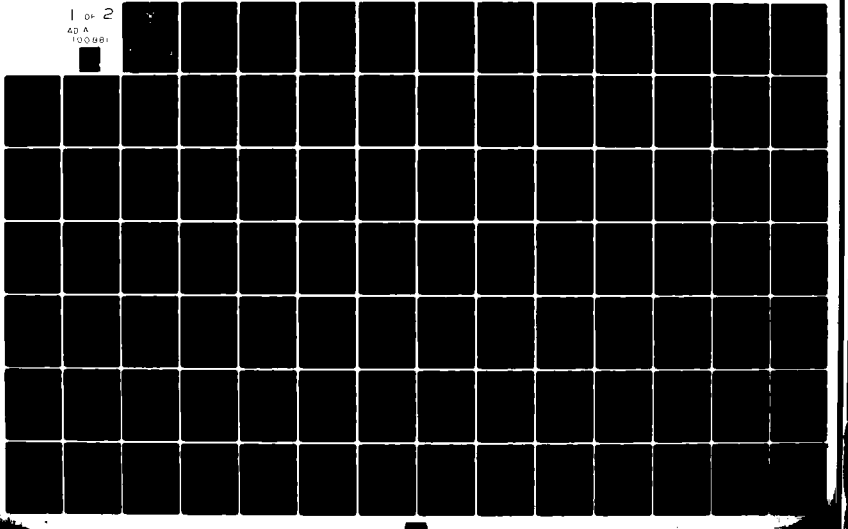
AD-A100 881 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOO--ETC F/G 9/2
JOVIAL (J73) TO ADA TRANSLATOR SYSTEM.(U)
DEC 80 R L BROZOVIC
UNCLASSIFIED AFIT/GCS/EE/80D-5 NL

1 OF 2
AD A
100881



DDC

LEVEL II

AD A100881

DTIC FILE COPY

DEC 1981

D

A

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY (ATC)

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

81 6 30 071

JOVIAL(J73) TO ADA

TRANSLATOR SYSTEM

THESIS

AFIT/GCS/EE/80D-5 Richard L. Brozovic
Capt USAF

AFIT/GCS/EE/80D-5

JOVIAL(J73) TO ADA TRANSLATOR SYSTEM

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air Training Command
in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

by

Richard L. Brozovic, B.S.
Capt USAF
Graduate Computer Systems

December 1980

Preface

I have been interested in programming languages since my first courses in computer science at the undergraduate level. With the introduction of Ada, I immediately became interested in the future of Ada and the development of software tools to support the Ada environment. The design and development of a system to translate Jovial(J73) into Ada provided me with an excellent opportunity to exercise some of the compiler techniques that I have studied and also explore some of the concepts of language translation. I also had the opportunity to learn another language, J73, and become involved with the identification of deficiencies in the still developing compiler. All considered, I enjoyed the project immensely.

During the design and development of this translation system, I received help from many sources. The Avionics Laboratory of the Air Force Wright Aeronautical Laboratories (AFWAL/AA) sponsored this project and several people deserve my thanks. Major Dan Burton was the project sponsor and provided considerable guidance and information regarding J73. Mr Mike Burlakoff was also of great help in learning J73. Many other people within the Avionics Laboratory also provided help from time to time. The sponsorship and support of the AFWAL/AA were greatly appreciated.

I would also like to thank my thesis advisor, Lt Col Jim Rutledge, who provided technical support on Ada. Major

Mike Wirth also deserves my thanks for his assistance in the areas of formal language theory. Thanks also goes out to many other AFIT instructors and fellow students who provided insight and guidance during the design and development of the project.

Finally, there are two other persons that deserve a special thanks for their part in my success. My wife Pat shared the good times with me and was there to support me through the hard times. Her encouragement was invaluable during my entire time at AFIT. The other person of great importance to me is my son, Jason. Although he did not understand why I could not always give him the attention he wanted, when we did spend time together, we enjoyed life greatly.

Contents

List of Figures

List of Tables

Abstract

In recent years the Air Force has developed a standard programming language, J73, for use in embedded computer systems. Now that the Air Force has a considerable investment in systems that are currently being developed with J73, the Department of Defense has selected a high order programming language, Ada, that will become the standard for programming embedded computer systems throughout the Department of Defense. Also under development is the definition of a support environment for Ada. One of the tools of this support environment should be a translator that will produce Ada source programs from J73 source programs (Ref 4).

The subject of this research project was the design and development of a Jovial(J73) to Ada Translator System (JATS). The resulting system accepts J73 programs and produces equivalent Ada programs to the extent possible while identifying segments of J73 code that were not translated. This project made use of language parsing techniques and various data structures to support the translation process. Several problem areas are identified with possible solutions. The translator system should be a useful tool in the transition from J73 to Ada.

I Introduction

In recent years the Air Force and Department of Defense (DoD) have initiated efforts to adopt standard programming languages in order to reduce the cost of software development for embedded computer systems. Embedded computer systems are an integral part of a larger weapon system, such as a flight control computer in an aircraft. Some of these embedded systems are so unique and specialized that none of the programming languages approved for use in Air Force systems have been appropriate for programming these systems. As a result, new programming languages have been introduced which compounds the problems of software development. In an effort to reduce software development costs the Air Force has developed J73 which is a derivative of Jovial, Jules' Own Version of the International Algebraic Language (Ref 1:39). As a result, the Air Force has a considerable investment in systems that are currently under development with J73.

In a somewhat later effort, the DoD also initiated the development of a DoD standard high order language for programming embedded computer systems. In April 1979, DoD selected Ada as the language and in July 1980 released a proposed standard for the language. In the near future, Ada is supposed to replace J73 on the list of approved programming languages for embedded computer systems (Ref 13). Also currently under design is the definition of

a support environment for Ada which should include, among other software tools, translators for present languages (Ref 4).

Although direct translation of existing software may not produce production quality software, a translator which performs the bulk of the translation can still be a valuable aid in the transition to a new programming language. It can save valuable time by reducing the amount of hand translation that is required and identifying those areas that do require manual translation. The Avionics Laboratory of the Air Force Wright Aeronautical Laboratories (AFWAL/AA) has a requirement for a J73 to Ada translation system which should directly translate, to the extent possible, J73 programs into an equivalent Ada program and identify those segments of code that were not fully translated.

The purpose of this research project was to design and develop a translator system that would produce equivalent Ada programs from J73 programs. The system, called JATS for Jovial(J73) to Ada Translation System, is a software product which was implemented on the AFWAL/AA DecSystem-10 and produces Ada text files.

The majority of the work in producing JATS was in the software design and development. There are three functional units of JATS: (1) parser module, (2) translation module, and (3) prettyprint module. The parser module processes the input language and constructs a parse tree, an internal

representation, of the program. The translation module manipulates the parse tree to construct an equivalent Ada program. The prettyprint module uses the results of the translation to produce a formatted text file.

The parser was developed from a software package from Lawrence Livermore National Laboratory (LLNL) (Ref 12). This package allows the user to develop a table-driven parser from a specification of the language syntax with which the user can develop a language processor. For this project, the parser is used to construct a parse tree, a tree structure representing the input program.

The translation routines process the parse tree and produce a modified tree that represents an equivalent Ada program. Finally, the prettyprint routine produces a formatted text file using the suggested formatting conventions in the Ada reference manual as much as possible.

The remainder of this document describes the various parts of the design and development of JAIS. Chapter II defines the requirements specified by the research project sponsor, AFWAL/AA. Chapter III presents the systems analysis that was performed prior to development. Chapter IV describes the software development. Chapter V presents the results of this project and recommendations for further work. Several appendices provide additional information for the interested reader. Appendix A briefly describes the LLNL software package. Appendix B defines the syntax of the

J73 language and identifies those language productions that have been translated. Appendix C contains a description of JATS and was designed to be a system maintenance aid. Appendix D is a user's guide and Appendix E is an example J73 program and its translated Ada version.

In the remaining discussions, for the purpose of clarity, reserved words and specific data names will be capitalized and will be discernable from context. Nonspecific elements of the J73 language will be enclosed in angle brackets. For example, <name> represents any valid J73 identifier. The elements of J73 correspond, as closely as possible, to the definition of the language in MIL-STD-1589A.

II Requirements Definition

In recent years, the high costs of software development have increased the importance of the use of software development tools. Currently, the USAF has a considerable investment in major software systems that are being developed with J73. With the introduction of Ada, which is supposed to replace J73 as the USAF's approved programming language for embedded computer systems, it has become evident that an automated translation system from J73 to Ada is needed.

The Air Force Wright Aeronautical Laboratories/Avionics Laboratory has stated a requirement for the development of such a system. Although the general requirement of translation is rather straightforward, it is no simple task. As pointed out in the development of the Autopsy project at the University of Southern California, "there are two constraints guiding the development of a useful translation system:

1. A translation system must do almost all of the work.
2. No translation system can do all of the work" (Ref 2:118).

Invariably there will be sections of code which cannot be directly translated because of unique system dependencies. In such cases it is very difficult to insure the correctness of the translation and the user must perform the translation

manually.

The general requirement can be divided into three specific parts. First, the translation system should accept any correct J73 program and produce, to the extent possible, an equivalent Ada program. The assumption that the input is syntactically correct will allow for the elimination of lengthy error detection and recovery routines. Secondly, when a direct translation is not possible, the translator should identify the untranslated code to allow the user to perform a manual translation. Thirdly, to insure portability of the translation system, its design should conform to MIL-STD-1589A, which defines the syntax and semantics of J73.

Since J73 is still undergoing changes, the system should be designed to allow modification of the system to meet the changing language environment. Since the scope of this research project was limited to translating a subset of J73, the system should also be designed to facilitate the addition of the remaining translation routines to complete the translation system.

In summary, JATS should translate J73 source programs into Ada source programs, identifying those segments of code that have not been translated. JATS should also be designed to allow maximum portability, meet changing langauge requirements, and allow completion of the translation process.

III System Analysis

The analysis of any large software project is an important task that should not be overlooked. One of the goals of any system analysis should be to identify all software development tools that may be applicable to the project. With increased emphasis on reducing software development costs, the use of available software that has been tested and implemented must be considered and can reduce the overall effort and therefore the costs of a development project. This is especially important in a research project which must be limited in scope in order to achieve a reasonable goal.

Translation of the entire J73 language was not possible within the time constraints of this project. Therefore a subset of the J73 language was selected for translation. This subset was chosen based upon the more commonly used features of the language so that the results would be applicable over as large a range as possible.

Although the project was limited to translating a subset of J73, the design of the system proceeded as if the entire J73 language was to be translated. This will allow the addition of the remaining translation routines with minimum impact on the rest of the system. Therefore, the design of the front end of the translator system, the parser, was a critical part of the project. There are two general techniques used in the development of parsers. One

is the use of top-down recursive descent parsers and the other employs table-driven bottom-up parsers. The first technique implements the parsing algorithm as recursive procedures whose algorithms depend upon the language to be processed. As a result, a general parser for more than one language cannot be developed since the parsing algorithms will change from one language to another. In contrast, the second technique uses algorithms that are language independent to generate and then use a set of parser tables. Since, the tables contain the specific information about the language, the algorithms can be implemented to generate parsers for any members of a given family of languages. The particular family of languages depends upon the algorithms used to generate the parser. One such parser generator package is "LR: Automatic Parser Generator and LR(1) Parser" designed at Lawrence Livermore National Laboratory (LLNL) (Ref 12).

LR Parser Generator

The LR package is a set of programs written in ANSI standard FORTRAN 66 which have been successfully used in many applications. The package is currently available on the Aeronautical Systems Division's Control Data Corporation CYBER 175 computer system and has been used in previous AFIT research efforts. The AFWAL/AA DecSystem-10 computer system can also support LR. The first program of the LR package, LRS, is the table generator. It is complete in itself but

has two implementation dependent subroutines that require minor changes to implement it on the DecSystem-10. The second program, PRS, is the parser skeleton and must be supplied with the necessary support routines to complete the language processor. These required routines are the lexical processor, which recognizes the lexical elements of the language, and the semantic processor, which defines the semantics of the language processing.

Since both parsing techniques require the lexical and semantic processors, the two techniques do not differ in that aspect. However, since a parser generator package was available, the table-driven bottom-up parser was chosen for this research project.

Decisions

Since LR was available for use in the development of a translator system, the major decision that had to be made before development could proceed was the choice of a programming language. Since LR was written in ANSI standard FORTRAN 66, the continued use of FORTRAN was possible, but undesireable. The main reason for this was the anticipated use of various data structures and recursive algorithms which cannot be implemented in FORTRAN. Also, one of the goals of the Ada support environment is to develop the software tools using Ada. Therefore, Ada would have been the obvious choice. However, the lack of a compiler would

have seriously restricted the development and testing of the system. The next choice was J73, which was an excellent choice for several reasons. J73 is a block structured programming language which allows the use of structured programming techniques and any system that supports J73 should also be able to support the translator system. Another possible reason might be the fact that the translator could then translate itself into an Ada version. However, this approach would not be equivalent to developing the translator in Ada.

There was only one drawback to designing the translator system in another language; the parser skeleton, PRS, was written in FORTRAN. However, this was a minor drawback since the parser skeleton was also available in a PASCAL version as a result of an AFIT class project for the Advanced Compiler Theory course. Since the algorithms used in the parser skeleton are short and straightforward, and were already implemented in a block structured programming language, the process of translating the parser skeleton into J73 did not involve much effort.

IV System Software Development

The bulk of this research project was the development of the Jovial(J73) to Ada Translation System (JATS) software. JATS consists of three functional modules: (1) parser module, (2) translation module, and (3) prettyprint module. The parser module was developed using the LLNL parser generator package and produces a parse tree of the source program. The translation module uses the parse tree and performs a direct translation of the nodes of the tree. The result of the translation is a modified parse tree which represents an equivalent Ada program. The prettyprint module uses the modified parse tree to build a formatted text file which contains the Ada program. Each module of the system will be described in detail.

Parser Module

The parser module implements a table-driven bottom-up parser designed using the LR parser generator. LR is a language processing tool consisting of two programs. The first program, LRS, is an automatic parser table generator. The second program, PRS, is a parser skeleton. When the tables from LRS, the lexical processor, and the semantic processor are added to PRS the result is an LR(1) parser and language processor. The LR(1) characteristic of the language is a design constraint of the grammar which defines the syntactic type of the language and means that the

language can be parsed from left-to-right with a lookahead requirement of one symbol. A language of this type can be recognized by a deterministic pushdown automaton in linear time (Ref 7:501). A deterministic pushdown automaton uses a stack and the knowledge of the current state and lookahead symbol to decide the next course of action. The parser tables from LRS provide the control information for the finite state automaton while PRS provides the definition of the process. The design of the parser module involved four major phases: (1) parser table generation, (2) parser table translation, (3) parser implementation, and (4) semantic processing. Each of these sections will be discussed in further detail.

Parser Table Generation. LRS accepts an LR(1) grammar in a modified Backus-Naur Form and produces a grammar analysis and a set of parser tables. However, some work was necessary to implement LRS on the DecSystem-10. LRS contains two subroutines that are implementation dependent, INIT and CHRIND. INIT initializes the input and output files. CHRIND converts a single character stored left-justified in one word into an integer, the character code, and required the use of a macro subroutine. This macro routine is a function called ICHR. With these two changes, LRS was successfully running on the DecSystem-10. Also, before LRS would process the complete J73 grammar, several other changes had to be made within it. Considering the size of LRS, this was no minor task. These changes

involved increasing the sizes of some of the dimensioned variables to accommodate the large grammar. These changes are discussed in greater detail in Appendix A.

One other change was made to enhance the use of LRS. In the version available at AFIT, both the grammar analysis and the parser tables were output to the same file and the parser tables had to be copied to another file with the use of an editor program. The grammar analysis consists of large amounts of data describing the parser and is usually output to the printer while the data tables are placed into a data file to be inserted into the parser skeleton. To separate these output files, one additional output subroutine was added to write the data tables to a different file. A description of this subroutine is contained in Appendix A.

The J73 grammar is input to LRS in a modified Backus-Naur Form, which is a convenient way of describing the syntax of programming languages. Several problems were encountered during this phase of the project since a complete LR(1) grammar was not available. After considerable effort to rewrite the grammar from MIL-STD-1589A into a form that was LR(1), an LR(1) grammar for the executable statements of J73 was obtained from Softech Incorporated (Ref 8). However, the remaining grammar for the declarations was not LR(1). According to Softech Inc., they developed the J73 compiler using a top-down

recursive descent parser for the declarations and a table-driven bottom-up parser for the statements and therefore did not have a complete LR(1) grammar available. However, according to the language designers at Softech Inc., J73 is a deterministic context free language, which implies (Ref 7:512) that an LR(1) grammar does exist for the language. Rewriting the grammar was no minor task and there are still several places in the grammar where the translator will not accept a syntactically correct J73 program. The resulting grammar is contained in Appendix B. The greatest discrepancy is probably the complete lack of a <block-preset> definition. The definitions of the <block-preset> and the <table-preset> conflicted, resulting in a grammar that was not LR(1). The grammar for the <block-preset> could have been rewritten but would have been a time consuming process. The <block-preset> was eliminated in favor of allowing development to proceed on the remaining modules of the system. However, use of the <block-preset> does not appear to be that extensive and this loss of completeness was assumed to be justified. Other discrepancies are concerned with some of the size constraints in some of the data declarations. The problem was that the <formula> in some of these cases was immediately followed by a left parenthesis while the grammar for a subscripted variable can be a <name> followed by a left parenthesis. For a grammar to be LR(1), the parser has to be able to decide what to do in its present state based

upon the next token, or symbol, of the input stream. However, with this conflict, the parser could not decide whether to combine the <name> and left parenthesis to generate the subscripted variable or reduce the <name> to a <formula> to generate, for example, a <status-size>. The specific areas that are constrained are the <status-size> in the declaration of a status item, the <bits-per-entry> and <entry-size> in the declaration of structured tables, and the <repetition-count> in a <table-preset>. Again, these constraints do not appear to place a serious limitation upon J73 users. These discrepancies are summarized in table I.

Table I

Grammar Discrepancies

MIL-STD-1589A Reference	Element	Current Definition
2.1.1.6	<status-size>	<integer-literal>
2.1.2.2	<bits-per-entry>	<integer-literal> <name> <function-call>
2.1.2.4	<entry-size>	<integer-literal> <name> <function-call>
2.1.6	<repetition-count>	<integer-literal>
2.1.6	<block-preset>	Not Defined

It should also be noted that a <name> which has not yet been defined during parsing of declarations is scanned as a

<name> while an undefined <name> during parsing of statements is scanned as a <proc-label-name>. This distinction was necessary in the grammar definition to make it LR(1).

Once an LR(1) grammar for J73 was defined, LRS produced a valid set of parser tables. The J73 grammar that was used to generate these parser tables is contained in Appendix B. The specific contents and structure of the tables are described in Appendix A.

Parser Table Translation. Once the parser tables were obtained from LRS, a translation step was required to transform the data tables into the J73 language in a form that was equivalent to the FORTRAN 66 DIMENSION and DATA statements. The data tables were processed by TABLES.J73 and produced J73 constant table declarations that duplicated the FORTRAN 66 data structures. These statements were then collected together as a compool module. J73 compool modules are separately compilable modules that contain data declarations. The use of the compool module proved to be very advantageous since the tables were very large and required about five minutes of cpu time for compilation on the DecSystem-10. The J73 compool module provided the parser table information for the parser skeleton.

Parser Implementation. PRS provided a FORTRAN parser skeleton which had to be translated into J73. This step was relatively simple since PRS had already been translated into

PASCAL and the same data structures were available in JTS. Four subroutines make up the main parsing algorithm: (1) FIND'REDUCTion, (2) FIND'TRANsition, (3) DO'REDUCTion, and (4) DO'TRANsition. FIND'REDUCT uses the current state and token to search the tables for a possible reduction. If one is found, the applicable language production is used by DO'REDUCT to call the semantic processor and reduce the information in the stacks accordingly. If a reduction is not possible from the current state, FIND'TRAN searches the tables for a possible transition to another state. If a transition is possible, DO'TRAN stacks the current information on the appropriate stacks and calls the lexical processor for the next token in the input stream. If neither a reduction nor transition is possible, the parser has detected an error and must recover. Since this translator was designed with the assumption that the input programs are correct, any errors detected will result from grammar discrepancies and that part of the program cannot be processed.

The major addition to the skeleton to complete the parser was the lexical processor or scanner. This subroutine performs a lexical analysis of the input and supplies the parser with the tokens of the input stream. For example, the assignment statement in figure 1 consists of three lexical elements: (1) TEST'FLAG which is a JTS identifier or <name>, (2) the equal sign, and (3) the reserved word FALSE. The lexical structure differs from one

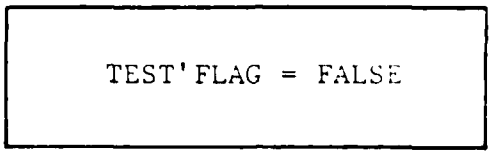


Fig 1. Assignment Statement

language to another implying the need for a specialized scanner. The lexical elements of J73 consist of: (1) reserved words and symbols, (2) identifiers or <names>, (3) numbers, (4) character strings or <character-literals>, (5) directives, and (6) comments. Procedures that are local to the lexical processor are used to scan the different elements of the language. These procedures as well as several functions which support the lexical processor are described in Appendix C.

Semantic Processing. The semantic processor describes the actual language processing. Without the semantic processor, the parser is complete in that it will recognize the grammar. However, for a language processor to perform a useful function, the definition of the semantic process must be provided. These routines are very dependent on the source language and the purpose of the language processor. For example, the semantic processor of a compiler typically contains the symbol table manipulator and code generator. However, for a translator, the semantic processor needs to preserve the structure of the source program until it has been completely parsed, at which time translation can

proceed. The data structure that was selected to represent the program was the parse tree. An example of a partial parse tree of the assignment statement from figure 1 is shown in figure 2. The labels on the links in figure 2 will be discussed in the next paragraph.

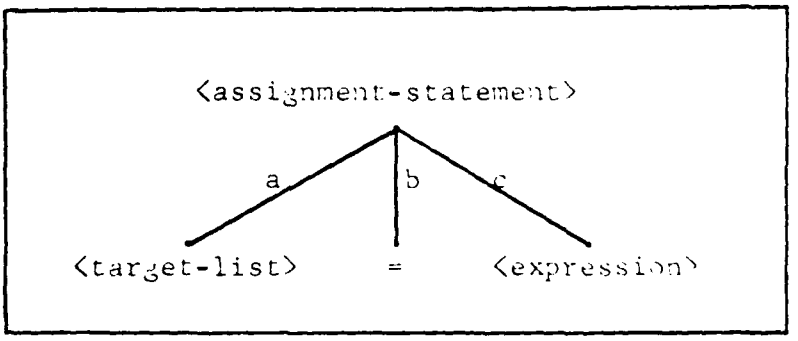


Fig 2. Partial Parse Tree

The reason that this structure was selected was because it records the complete parse of the source program and allows further processing on an element-by-element basis.

The specific data structure that was selected to represent the parse tree was a linked list. The node which represents the assignment statement in figure 2 would consist of four list elements, one describing the node and three identifying the subtrees of the node, and is shown in figure 3. This is accomplished by using four identical elements each containing the three following fields: (1) NODE'PTR which is an integer field, (2) LINK'TYPE which is an integer field, and (3) NODE'LINK which is a pointer field containing an absolute machine address. The contents

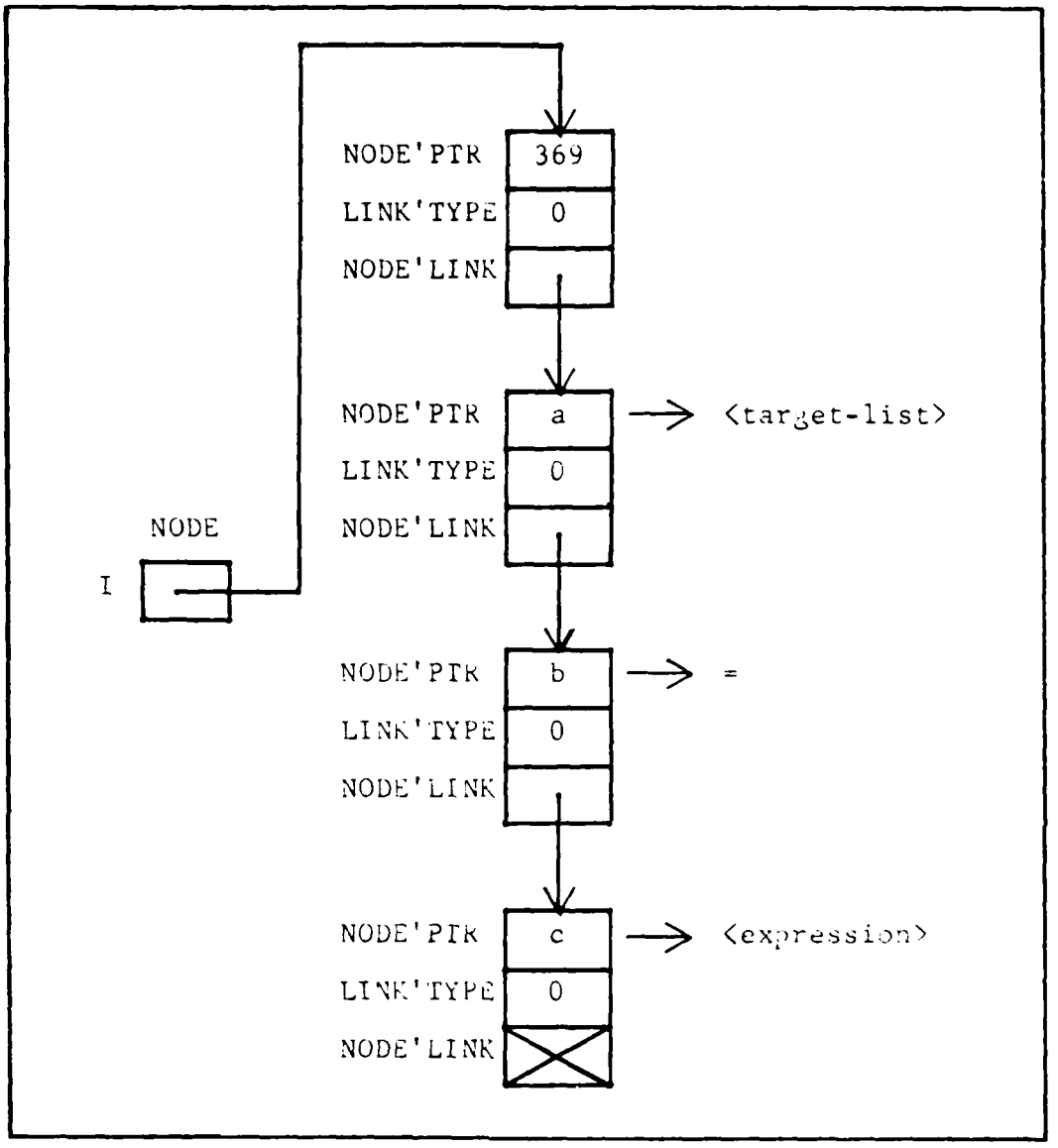


Fig 3. <assignment-stmt> Node

of NODE'PTR of the first element of the linked list representation of a parse tree node contains the production number, or language rule, that was used to obtain the node from its subtrees. NODE'PTR of all other elements of a node contain the node number which identifies the subtrees. The

values contained in NODE'PTR of these remaining elements correspond to the links shown in figure 1. LINK'TYPE = 0 for all elements of an internal node of the tree and indicate that NODE'LINK points to the next element of the node. The final element of a node will have a NODE'LINK that has a null value. Using this structure to describe a node within the tree, the nodes of various sizes can be efficiently stored without wasted space, a very important consideration in such a large software project. This structure also allows for the dynamic nature of the tree during the translation process.

If the node is a terminal node, a leaf of the tree, then the value of LINK'TYPE will be non-zero and the other fields will have other meanings. For all terminal nodes, NODE'PTR will contain the token number of the terminal symbol. If the terminal node is a reserved word or symbol, LINK'TYPE will have the value 3, and NODE'LINK will have a null value. If the terminal node is a <name> or <character-literal>, LINK'TYPE will have the value 3, but NODE'LINK will point to another element which will be interpreted as follows. If the node is a <name>, NODE'PTR = 0, LINK'TYPE = 3, and NODE'LINK points to the symbol table entry for the <name>. If the node is a <character-literal>, NODE'PTR contains the index into the string table to identify the string, LINK'TYPE = 3, and NODE'LINK has a null value.

If the terminal node is an <integer-literal>, LINK'TYPE = 1, and NODE'LINK points to an element that contains the integer value of the node. If the terminal node is a <real-literal>, LINK'TYPE = 2, and NODE'LINK points to an element that contains the real value of the node.

The parse tree is a collection of these nodes, such that the root of the parse tree is the <complete-program> and NODE(I) contains a pointer to the first element of the Ith node. J73 has pointer variables which can be easily manipulated to implement a linked list structure; however, the current compiler does not yet support execution time or dynamic allocation of memory. Therefore, it was necessary to declare a HEAP, or large storage space, which can be used to store these dynamic elements. In order to make efficient use of such a heap, a dynamic allocation algorithm, implemented as a function NEWP, is used to control allocation of new elements. During the parsing process the parse tree is continually growing and deallocation of used heap space does not occur. Therefore, NEXT'NODE, which is the index of the next available NODE, needs only to be incremented by one each time a node is allocated. Likewise, NEXT'HEAP, which is the pointer address of the next available heap space, needs only to be incremented by the size of the space to be allocated. Additional routines and data structures to implement the allocation algorithms will be covered during the discussion of the translation module.

The heap is also used for storage of the symbol table. For a translation system, the entire symbol table must be kept until the translated program can be output since the terminal nodes simply contain a pointer into the symbol table to identify the <name> of the node. This is contrasted to the maintenance of the symbol table in a one pass compiler where the symbol table entries for a given lexical level can be deleted when that lexical level is exited. The symbol table is stored as a hash-coded collection of separate chains (Ref 10:513) where HEAD(I) contains a pointer to the first symbol table entry in the Ith chain. Each element of the symbol table contains five fields: (1) SYM'STORE'PTR which is the index into the character storage array for the first character of the symbol, (2) SYMBOL'LENGTH which is the number of characters in the symbol, (3) SYM'LINK which contains a pointer to the next element of the chain, (4) TYPE'CLASS which is used to define the class of the symbol, and (5) PROC'NUM which indicates which procedure the symbol was declared in. The possible type classes are listed in table II and are used during the parsing process to return the proper token symbol from the lexical scanner. PROC'NUM is used to allow the same <name> to be used at different lexical levels. Another array, PARENT'PROC, is used to identify the nesting of the procedures.

Symbol table entries are made by the scanning procedures at the time that the symbol is first scanned in

TABLE II

Type Classes

Value	Classification
0	<name>
1	<item-type-name>
2	<table-type-name>
3	<block-type-name>
4	<proc-name>
5	<label-name>
6	<define-name>
7	<proc-label-name>

procedure NAME'RESWORD. The scanner also builds the terminal nodes of the parse tree and provides the number of the node to the semantic processor for the semantic stack. The remainder of the semantic processing is accomplished in procedure SEMANTIC, which is called from DO'REDUCT when a reduction is to take place. The semantic stack will contain the node numbers of each of the subtrees of the current production, or rule, that is to be reduced. The semantic process consists of building a new node which combines the subtrees of the production. For example, the production used to build the assignment statement is shown in figure 4.

```
<assignment-stmt> ::= <target-list> = <expression>
```

Fig 4. Assignment Statement Production

When SEMANTIC is called, the number of this production is passed as a parameter and the new node is built from the

semantic stack. The elements on the right hand side of the production are at the top of the stack and are used to construct a new node as follows: (1) the production number is used to construct the first element of the new node, (2) the last element of the node is on top of the stack and is added to the node by creating a new element and linking it to the first element. The process of popping the top element from the stack and inserting it after the first element of the new node is repeated until the complete node has been constructed. The reduction process then pushes the new lefthand side of the production and the new node that was just constructed onto the stacks.

The semantic routines also add information to the symbol table when a declaration is processed to establish the class of the symbol table entry. For example, the production in figure 5 would cause the symbol table entry for <name> to be set to indicate that the entry is a <item-type-name>.

```
<item-type-dec> ::= TYPE <name> <item-type-description> ;
```

Fig 5. Declaration of Item Type

The same actions would apply for processing of block declarations, procedure declarations, label declarations, table declarations, and define declarations.

When the parsing module has completed its processing the parse tree and the symbol table will have been constructed to allow the translation process to begin.

Translation Module

Translation of the J73 program is accomplished in two separate steps. The symbol table is processed first followed by the processing of the parse tree. Processing of the symbol table translates J73 identifiers into Ada identifers. The identifiers of both languages consist primarily of letters and digits. The only differences between the identifiers is the use of the dollar sign and the single quote in J73 and the use of the underscore in Ada. The dollar sign is used to represent special characters that may be required in external names and is implementation defined. Ada has no equivalent structure; therefore, the dollar sign is not translatable and is not allowed in programs to be translated. The use of the J73 single quote is similar to the use of the underscore in Ada but J73 allows a more liberal use of the single quote. After surveying the programmers who use J73, it was found that the most common usage of the single quote was as a seperator between parts of an identifier. Such usage conforms to the rules of the use of the underscore in Ada and the translator was therefore designed with the requirement that the use of the single quote conform to the rules of the use of the underscore in Ada. One other

assumption was made to preclude the requirement that JATS generate new identifiers: J73 identifiers will not consist of Ada reserved words. This greatly simplified the translation of identifiers since all that is now required is changing all single quotes into underscores during a single pass through the symbol table character string storage. Processing of the symbol table in this manner results in each identifier being translated exactly once instead of processing an identifier each time that it is found in the parse tree. If JATS was an interactive system, the above limitations could be removed since JATS could request assistance from the user to rename any identifier that was found to not conform to the rules of Ada.

The remainder of the translation process consists of manipulation of the parse tree by procedure TRANSLATE. Translation of the terminal nodes, or leaves, is not required since the translation of identifiers has already been accomplished and blanket translation of reserved words and symbols is not performed. The translation process is language production oriented, that is, each node is processed based on the production number that was used to construct the node. The tree is processed in a bottom up fashion, with each subtree of a given node being processed before the node itself will be processed. Recall that the first element of each internal node contains the production number for the node and this is used to select the appropriate action through a case statement. Each language

production will have a case alternative that defines the specific actions, possibly none, that take place during translation of a node that was constructed with the production. There are four basic processes that are performed during translation: (1) changing reserved words and symbols, (2) adding new terminal symbols, (3) deleting existing elements, and (4) rearranging the elements of the node. Each of these operations will be discussed further.

The process of changing a reserved word or symbol is simple given the structure of the parse tree. Once the proper element of the node is identified by a pointer variable, NODE'PTR of that element identifies the terminal node that contains the reserved word or symbol. The NODE'PTR of this terminal node contains the token number that identifies the reserved word or symbol. The process of changing the reserved word or symbol then simply requires changing the value of NODE'PTR at the terminal node. The process of changing a symbol or reserved word is performed by procedure CHANGE'NODE. The translation process of the assignment statement from figure 2 is shown in figure 6. When the node that represents the <assignment-stmt> is translated, a pointer variable, N'PTR, is updated to identify the second element of the node. NODE'PTR at this element identifies the node that represents the equal sign. The value of NODE'PTR for the equal sign is 48 and is changed to a value of 171 to represent the assignment operator of Ada.

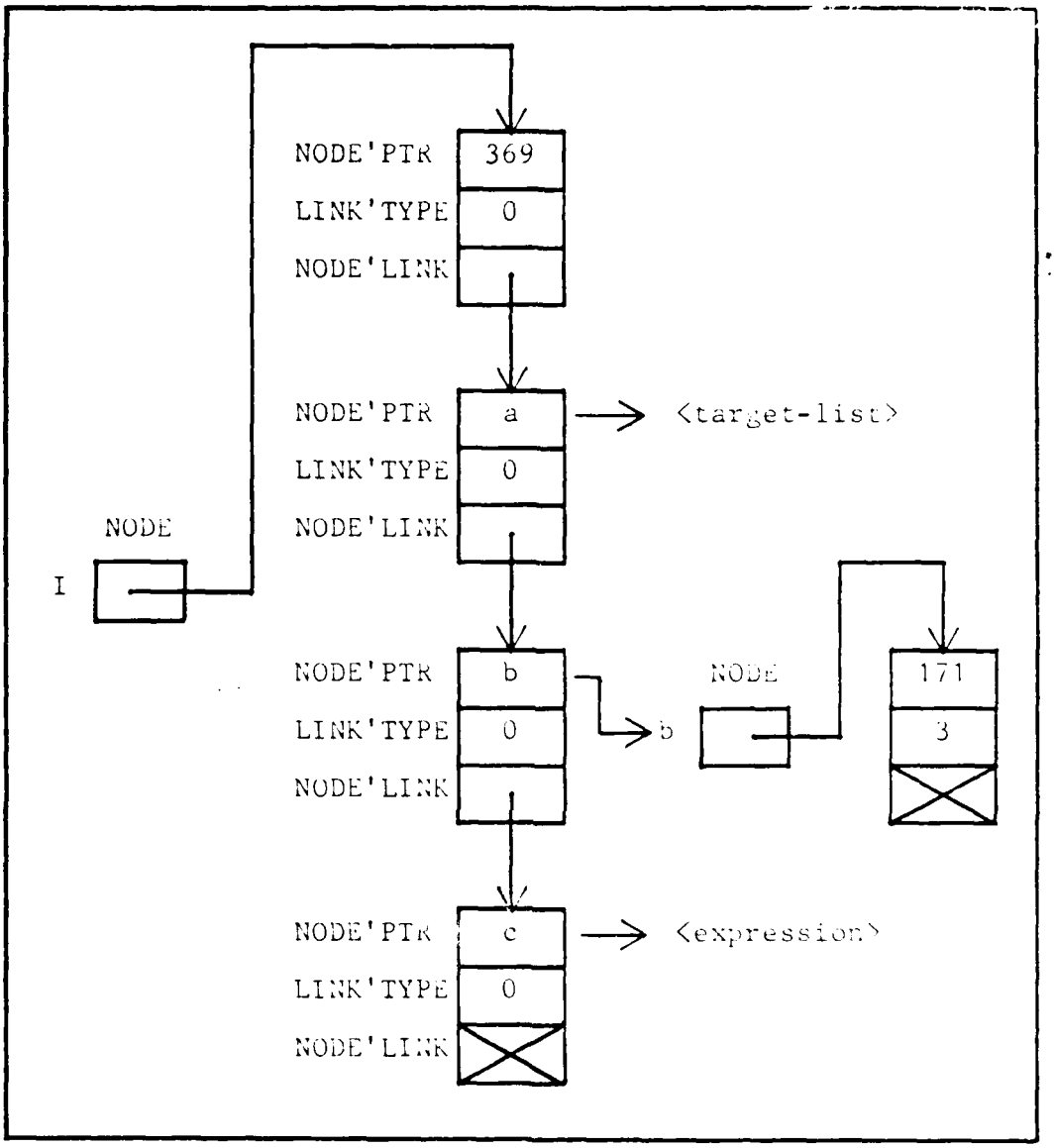


Fig 6. Translation of <assignment-stmt>

The process of adding a new terminal symbol to the production is a little more difficult. It involves creating a new node and inserting a new element in the chain of elements which makes up the node. This requires that the position of the new element be identified and the new

terminal symbol be specified. Once the element that will precede the new element has been identified, procedure ADD'NODE will create a new node that represents the new terminal symbol and establish the necessary links. This process is shown in figure 7 for the translation of a <label> where a new symbol, <<, must be added before the <proc-label-name> and the colon following the <proc-label-name> is changed to >>.

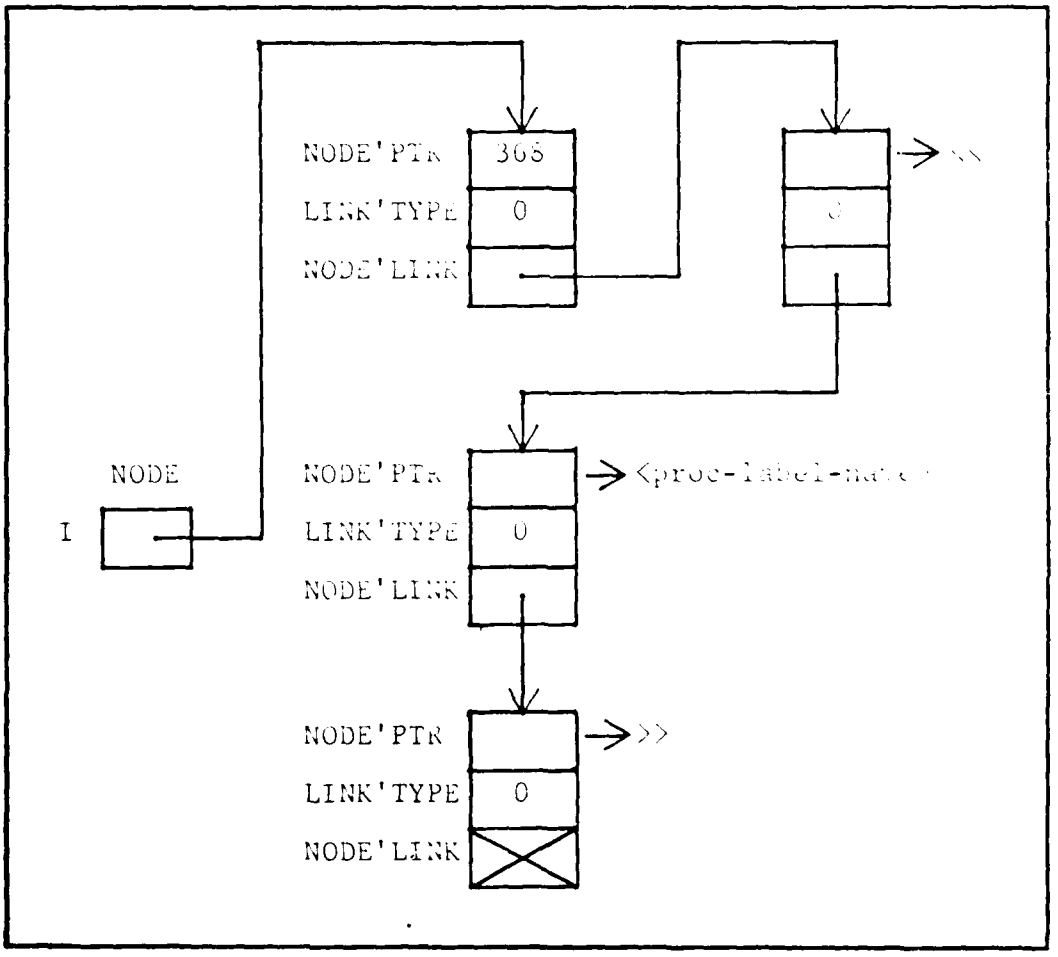


Fig 7. Translation of <label>

The process of deleting an element of a node is rather simple, just relink the appropriate elements of the chain to eliminate the desired element. However, since the tree is a dynamic structure, the utilization of the heap is most efficient when some type of storage reclamation is performed as elements of the tree are deleted. Therefore, two procedures, DELETE'NODE and DELETE'SUBTREE, are used to free heap space when a node element is deleted from the parse tree. DELETE'NODE releases the identified element from the node's chain while DELETE'SUBTREE releases all NODES and elements of the subtree that were deleted as a result of the deletion of the initial element. However, some changes to the allocation algorithms were required to implement reallocation of the space collected through storage reclamation. Using an overlayed data structure each node can contain either a pointer value, NODE(I), to a node chain or an integer value, FREE'NODE(I), which indexes the next node of the linked list of available nodes. When NODE(NEXT'NODE) is allocated, FREE'NODE(NEXT'NODE) contains a link to the next available node. Note that NEXT'NODE cannot simply be incremented as during the parsing process.

Storage reclamation of the released node elements is accomplished by linking the elements together using the already available pointer field NODE'LINK. If a list of available elements exists, FREE'PTR is a pointer to the first available element in the list and NODE'LINK at FREE'PTR is a link to the next available element. If the

list is empty, FREE'PTR = NULL, and allocation from the heap is necessary.

Rearranging the elements of a node is also a simple operation with the structure of the parse tree and corresponds to swapping the order of the subtrees of a node. Procedure SWAP'NODES is used to swap two elements that are adjacent. If the desired elements are not adjacent, repeated swaps must be performed until the desired change is achieved. An example of this process can be seen in the translation of a <program-body> using production number 27. The structure of the parsed node is shown in figure 8a and figure 8b shows the results after swapping the elements that correspond to the nodes defining BEGIN and <dec-list>.

```
a. BEGIN <dec-list> <stmt-list> <sub-def-list>
                                <direct-compound-end>

b. <dec-list> BEGIN <stmt-list> <sub-def-list>
                                <direct-compound-end>

c. <dec-list> BEGIN <sub-def-list> <stmt-list>
                                <direct-compound-end>

d. <dec-list> <sub-def-list> BEGIN <stmt-list>
                                <direct-compound-end> ;
```

Fig 8. Translation of <program-body>

Then <stmt-list> and <sub-def-list> are swapped resulting in figure 8c. Finally, BEGIN and <sub-def-list> are swapped and a semicolon is added as the last element of the node to

complete the translation as shown in figure 8d. Note that the first element of the two that are to be swapped must be identified by a pointer variable each time.

One other procedure is used during the translation process to simplify the actions taking place. That is the procedure MOVE'PTR, which simply moves the pointer in the node's chain of elements a specified number of elements.

Most of the language productions that have been translated used simple additions, deletions, and changes to the nodes. However, some of the translations required more work. One example is the translation of integer item declarations that specified the <integer-size> of the variable. Although the translation does not include the representation specifications of Ada, range constraints are placed on integer variables when possible. If the J73 integer was unsigned, then the range is specified as 0..2**<integer-size>-1. If the integer was signed, then the range of the resulting equivalent Ada variable is -(2**<integer-size>-1)..2**<integer-size>-1. However, the translation still uses simple additions to construct the node.

J73 character items are translated into Ada objects of type STRING, where the string would be indexed in the range 1..<character-size>. With this convention, all <character-literal>s can be translated into Ada strings.

In summary, the translation process involves changing, adding, deleting, or rearranging elements of the parse tree in order to have the parse tree represent an equivalent AED source program. Most of the translation routines simply required the application of one or more of the support procedures. It should also be noted that a good many of the translation routines required no action at all except for indicating that the node had been translated.

Since the scope of this research project could not encompass the entire translation process, JAFS currently translates only a subset of the J73 language. However, the design of the project included the complete translation process and additional translation processing can be added to the existing system without major modifications. In order to make the current translator as useful as possible, the subset of J73 that has been translated was selected to cover as much common usage of the language as possible. This includes: (1) basic program structure, (2) data item declarations and type declarations, (3) procedure definitions, and (4) most basic statements. A summary of the translated productions is contained in Appendix B.

To aid the user in identifying any sections of J73 code that may not have been completely translated, some method of marking each node as having been translated or not must be used. The technique used in this system is an array of boolean variables that correspond to each node of the tree.

The entire array is initialized with the value FALSE and as NODE(I) is translated the corresponding value in TRAN'FLAG(I) is set TRUE. This will then be used during the prettyprint process to identify the sections of code that were not translated.

Prettyprint Module

The usefulness of an automated translator system would be questionable if the format of the output was difficult to interpret. In order to get the most out of the translation, the resulting program should display the structure of the language as much as possible. The Ada reference manual suggests some formatting conventions to display the structure of the language. The prettyprint module outputs the translated program using these formatting conventions. The lexical elements that will make up the Ada program are the leaves of the translated parse tree and are output from left to right. Indentation and line breaks are controlled through the use of a case statement looking for particular reserved words and symbols. For example, with the exception of a formal parameter list in a procedure definition, the semicolon is a statement terminator and is used to generate an end-of-line and carriage-return. Another example is the reserved word ELSE, which must be output two spaces to the left of the current tab marker while the statements between the ELSE and END IF are output at the tab marker.

Also output during the prettyprint process are warning messages that identify sections of code that have not been translated and require user translation. This allows the user to quickly identify those sections of code which will require attention.

Conclusion

JATS is a fairly complex language processor that consists of three functional modules. The parser module which was developed using the LLNL software package constructs a parse tree of the J73 program. The translation module modifies the parse tree to represent an equivalent Ada source program. Finally, the prettyprint module builds a formatted text file to allow the user to edit the translated source code.

However, the user should be aware of several limitations of JATS and some of the problems of direct syntax translation. These will be discussed in the next section. Still, JATS should prove valuable for introducing Ada to current J73 programmers.

V Results and Recommendations

The primary result of this research project is a translation system, JATS, which provides a limited capability for J73 users to automatically translate J73 programs into equivalent Ada programs. Any user must be cautioned on several points before using JATS. The most obvious caution is that JATS, as it is currently implemented, is incomplete and places several restrictions upon the user. These include: (1) use of the single quote in J73 declared <name>s must conform to the use of the underscore in Ada and the use of the dollar sign is not allowed, (2) J73 <name>s cannot be Ada reserved words, (3) <block-preset>s are not allowed in programs to be translated, (4) size specifiers in some data declarations have limited definitions as shown in table I, and (5) use of DEFINEs is not allow since the mechanism for processing the DEFINEs has not yet been incorporated in JATS. A summary of the translated J73 language productions is contained in Appendix B.

Another point that all users of JATS should be aware of is that JATS translates a J73 algorithm into an equivalent Ada algorithm. This means that, while the translated code will be correct, the translated version may not make use of the features of Ada. Such an example is the use of declared boolean flags to carry an error condition through the code. During the translation process, the code will be translated

correctly but Ada's exception handling techniques would probably be more appropriate. This defficiency is not unique to JATS but is an inherent problem in automatic translator systems.

With these limitations in mind, JATS was tested against small example J73 programs that primarily consisted of the translatable language constructs. In all tested cases, translation was correct where possible and those sections of code that were not translated were properly identified. One exception to this result is the WHEN OTHERS case alternative in the Ada version of the case statement. It will be the first alternative in the translated version rather than the final alternative of the case statement. An example J73 program and its translation is provided in Appendix E.

This research project not only produced a valuable tool to aid in the transition to Ada as the DoD standard programming language for embedded computer systems but also provided the author with an invaluable experience in the development of language processing software. A task of this magnitude presented a real challenge for the author in several areas: (1) grammar analysis, (2) parsing techniques, and (3) translation techniques.

The LR automatic parser generator package from LLNL has been used at AFIT in other language processors, mainly compilers and interpreters. This project added a translator to the list of projects developed using LR. It also

required modification of LRS to accept a larger input grammar, which will allow future users to process large grammars with LRS. However, it should be noted that LRS is a very large program with a very large memory and processing time requirement.

One other area that required considerable effort was designing an LR(1) grammar for J73. Initially, the J73 grammar was obtained from MIL-STD-1589A and required extensive modification to produce a grammar that was LR(1). After considerable effort to produce a fully LR(1) grammar, an LR(1) grammar for the executable statements of J73 was obtained from Softech Inc.; however, several problems still existed in the grammar for the declarative parts of J73 and, as a result, several limitations have necessarily been placed on the user.

The other major areas of effort were the parsing and translation processes and the use of the appropriate data structures for efficient utilization of resources and ease of translation. Although the data structures themselves were not new, the implementation of some these structures was a new experience for the author. An example of this is the dynamic allocation algorithms and the use of the OVERLAY declaration to allocate the same memory to two different data structures, the tree nodes and a linked list of available nodes.

The process of translation also identified several

areas that present some difficulties. The translation of bit variables will require real work. If the bit variable has a size of one bit, then it can be translated into a boolean variable, and the bit literals of '1' and '0' can be translated into the boolean literals TRUE and FALSE, respectively. However, if the bit variable is larger than one bit, then the translation must create an abstract data type that will allow the bit literals to be translated easily. Also required during translation of bit literals is consideration of the bead size. The bead size of a bit literal refers to the weight of each character of the literal string such that a bead size of 16 indicates that the string represents a hexadecimal value. The valid bead sizes range from one to sixteen. In other words, the octal bit literal 3B'24' is equivalent to the binary bit literal 1B'010100' and some convention of translating the bit literals must be established depending upon the specific data structure used to implement the bit variables.

The translation of parameter lists for subroutines will also be a serious problem. In J73, the typing of the parameters occurs during the processing of the declarations in the <subroutine-body>. However, Ada requires that the type of the parameters be included in the parameter list. Therefore, translation of parameter lists will require the generation of new type declarations from the declarations that are "hidden" further down the parse tree, so that the type names can be used in the parameter lists. Declaration

of some of the data items will also require generation of new type declarations, for example floating and fixed data items.

Other items that are not directly translatable are the directives. Some of them will translate to equivalent Ada pragmas but others have no counterpart in Ada. For example, the COPY directive corresponds to the INCLUDE pragma but the register directives have no equivalent pragmas in Ada.

One additional experience gained by the author was in the use of J73. Although experienced with several other block structured languages, J73 still presented some new experiences, especially since the language and compiler are still evolving. However, J73 was an enjoyable language to work with; the only real problem encountered was the lack of execution time error diagnostics, which sometimes made the debugging process difficult.

A project of this magnitude very often leaves room for future work, and several areas have been identified as candidates for continued development efforts. These recommendations are listed below.

1. Modify JATS to translate J73 as defined in MIL-STD-1589B. The current version translates MIL-STD-1589A and the B version was released too late to be included in this research project. However, JATS was designed to allow a redefinition of the input grammar without major

modifications to JATS. It should be noted that the source code for JATS itself currently conforms to MIL-STD-1589b.

2. Complete the translation module. As currently implemented, JATS is incomplete and cannot translate all of J73. The addition of the remaining translation routines will allow JATS to automatically translate more of the J73 syntax. JATS was designed to allow the addition of the missing translation routines.

3. Add the capability to process external files so that COMPOOL and COPY directives can be used in the J73 program. This will require the construction of the symbol table for the declarations that are contained in the compool module, and scanning of the external file for the COPY directive.

4. Improve the translation process by making it interactive and allow the user to identify sections of code that require manual translation. This would be possible using the currently available SKIP, BEGIN, and END directives. Then JATS could allow the user to specify the translated code interactively for the identified code and for any difficulties that the translator may encounter.

This research project incorporated the use of various techniques in several topic areas and resulted in a challenging and rewarding experience for the author. The author hopes that this project will generate continued

interest in the subject areas of Ada and automatic translator systems.

Bibliography

1. "Air Force Working on Single Language for its Computers," Electronics, 51: 39-40 (25 October 1978).

2. "1979 Annual Technical Report: A Research Program in Computer Technology," Report ISI/SR-80-17, Information Sciences Institute, University of Southern California, Marina del Rey, California, June 1979.

3. Defense Advanced Research Projects Agency. Requirements for Ada Programming Support Environments: Stoneman. Washington, D.C.: Department of Defense, February 1980.

4. Defense Advanced Research Projects Agency. Requirements for the Programming Environment for the Common High Order Language: Pebbleman. Washington, D.C.: Department of Defense, July 1978.

5. Fisher, David A. "DoD's Common Programming Language Effort," Computer, 11: 24-33 (March 1978).

6. Gillmann, Richard. "An Intermediate Language for CMS-2 and Ada," Autopsy Note 15, Information Sciences Institute, University of Southern California, Marina del Rey, California., 12 April 1979.

7. Harrison, Michael A. Introduction to Formal Language Theory. Reading, Massachusettes: Addison-Wesley Publishing Company, 1978.

8. J73 Statement Grammar. Softech Incorporated, Waltham, Massachusettes, August 1980.

9. Knuth, Donald E. The Art of Computer Programming, Volume 1, Fundamental Algorithms. Reading, Massachusettes: Addison-Wesley Publishing Company, 1973.

10. Knuth, Donald E. The Art of Computer Programming, Volume 3, Sorting and Searching. Reading, Massachusettes: Addison-Wesley Publishing Company, 1973.

11. MIL-STD-1589A. Military Standard Jovial (J73). Washington, D.C.: Department of Defense, 15 March 1979.

12. Wetherell, Charles and Alfred Shannon. LR: Automatic Parser Generator and LR(1) Parser. Report UCRL-82926. Lawrence Livermore National Laboratory, California, 14 June 1979.

15. Whitaker, William A. "Ada - The DoD Common High Order Language Effort," NAECON 1979, 3: 1272-1275 (March 1979).

Appendix A

LR: Automatic Parser Generator and LR(1) Parser

"LR is a pair of programs--an automatic parser generator and an LR(1) parser. The parser generator reads a context-free grammar in a modified BNF format and produces tables which describe an LR(1) parsing automaton. The parser is a suite of subroutines which interprete the tables to construct a parse of an input stream supplied by a (locally written) lexical analyzer. The entire system may be used to generate parsers for compilers, utility routines, command interpreters, and the like. LR and its predecessors have been in use at Lawrence Livermore [National] Laboratory (LL[N]L) for ten years. LR's outstanding characteristic is the ease with which new tables can be generated to reflect a change in the language to be parsed. This flexibility is prized by programmers writing utilities and command interpreters whose input languages typically grow and change during program development. LR is written entirely in ANSI standard FORTRAN 66 and requires only minor changes when moved to a new computer" (Ref 10:1).

LR was an important part of the JAIS software development. Major Michael Wirth has been instrumental in promoting the use of LR in the development of compilers and interpreters at AFIT. A complete understanding of LR requires knowledge of formal language theory but is not required to use the package. However, for the interested reader, an understanding of the data structures used in the parser tables is helpful in understanding the parsing process.

The parsing tables produced by LRS consist of a collection of arrays, FORTRAN dimensioned variables, that contain the control information for the finite state

automaton which performs the parsing.

The vocabulary of the language is described using two arrays, V and VOC. V contains a contiguous stream of character strings which are the vocabulary elements. VOC contains pointers into V to identify the start of the vocabulary string that has the token number which is the index into VOC. Figure 9 illustrates this relationship and shows that the token number for the reserved word PROGRAM is 138.

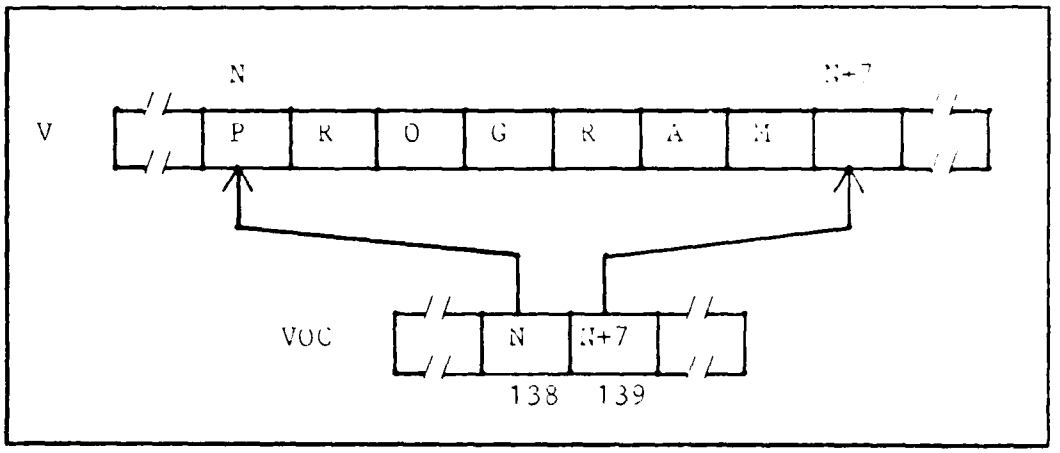


Fig 9. Vocabulary Data Structure

The transition data is contained in three arrays: FIRN, TRAN, and ENT. FIRN contains pointers into TRAN which contains a set of state numbers for possible transitions. ENT contains the token number of the required lookahead symbol for a transition to that state. Figure 10 illustrates this structure. Thus, for a given state, S, a

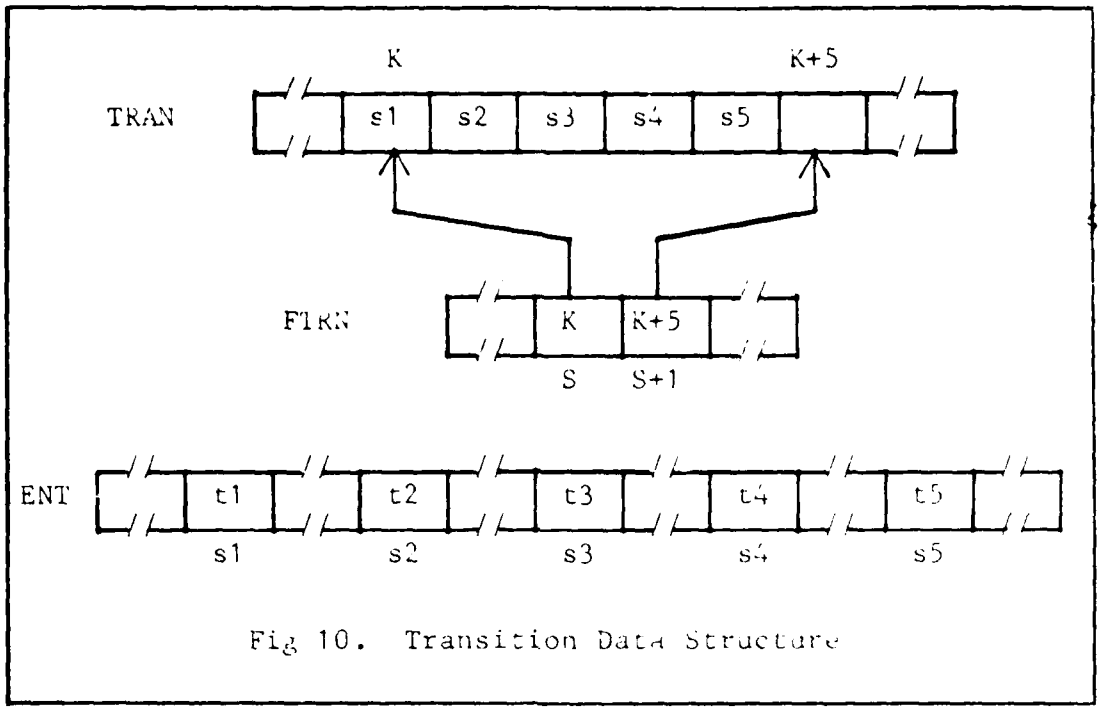


Fig 10. Transition Data Structure

search of the possible states in TRAN for an allowable transition based upon the current lookahead symbol and the entrance symbol will identify the next state of the automaton. Thus as shown in figure 10, from state S a transition to one of five states, S1,...,S5, is possible based upon the lookahead tokens T1,...,T5, respectively.

The reduction data is contained in seven arrays: FRED, NSET, LSET, LS, PROD, LHS, and LEN. LS and LSET define the lookahead sets where LS contains the lookahead sets as a contiguous array of integers, representing the token numbers, and LSET identifies the beginning of each lookahead set. Therefore, each lookahead set can be identified by its index into LSET. NSET contains collections of lookahead set

numbers and FRED contains pointers into NSET which identify the collection of lookahead sets which apply for the given reduction. If one of the lookahead sets of the given state, S, contains the current token, PROD(S) contains the language production number that will be used to perform the reduction. Figure 11 illustrates this relationship.

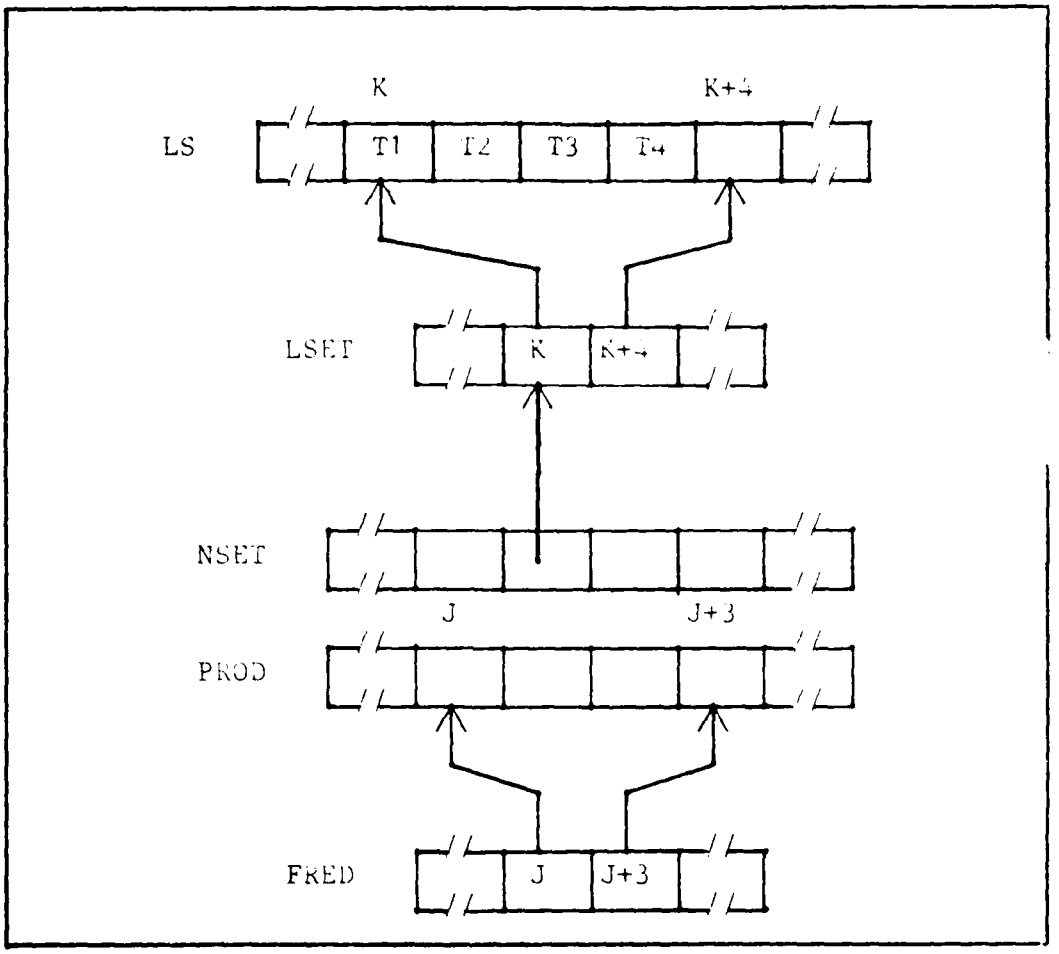


Fig 11. Reduction Data Structure

Thus, for a given state S, FRED identifies, through NSET and LSET, which lookahead sets must be checked to allow the

reduction in PROD to be used based on the current state and current lookahead symbol. The remaining two arrays contain the reduction information. LEN contains the length of the righthand side of the production and LHS contains the token number of the symbol on the lefthand side of the production. When the reduction is performed, LEN will indicate the number of elements that will be popped off of the stacks and LHS contains the token number that will be pushed onto the token stack.

Note, these data structures are produced with the FORTRAN 66 option for table structure and were retained in JATS in order to use the same parsing algorithms that were supplied in PRS.

However, before a complete set of parser tables was generated several changes were required to allow LALR to accept the large J73 grammar. These changes involved increasing the sizes of some of the dimensioned variables and other related variables. The string storage for the input grammar was limited by the dimension of SSTORE and the value of MAXSST. To allow for the large number of productions, the value of MXPROD and the dimension of PROD had to be increased. Also dependent upon the number of productions was the dimensioned variable MARK within subroutine COMPLT. The storage of reduction and transition data was limited by the dimensions of RED and TRAN, respectively, and the values of MAXRED and MAXTRN,

respectively. Other dimensioned variables used in the grammar analysis that had to be increased in size were BASIS, NEXT, ITEM, and PRODCN. These changes are summarized in table III.

TABLE III

Changes to Dimensioned Variables of LRS

Dimensioned Variable	Dimensions Old	Dimensions New	Related Variable
SSTORE	3000	10000	[illegible]
PRDIND	500	1000	[illegible]
TRAN	4000	26000	[illegible]
RED	4000	8000	[illegible]
NEXT	5000	8000	[illegible]
ITEM	5000	8000	[illegible]
BASIS	8000	24000	[illegible]
PRODCN	4000	6000	[illegible]
MARK	500	1000	[illegible]

The other change that was made to LRS was for convenience. Another subroutine, called OUTTAB, was added which created separate output files for the parser tables and grammar analysis. This change required that all calls to OUTPUT within subroutines TABLES and HLDDMP be changed to OUTTAB.

Appendix B

J73 Language Productions

The following list is the language productions output from the grammar analysis of LRS. Those productions that have been translated are preceeded by an asterisk.

```
  1 <SYSTEM GOAL SYMBOL> ::= END <COMPLETE-PROGRAM> END

 *2 <COMPLETE-PROGRAM> ::= <MODULE>
 *3                      / <COMPLETE-PROGRAM> <MODULE>

 *4 <MODULE> ::= <COMPOOL-MODULE>
 *5            / <PROC-MODULE>
 *6            / <MAIN-PROGRAM-MODULE>

  7 <COMPOOL-MODULE> ::= START <COMPOUND-COMPOOL> <NAME> ;
                         <COMPOOL-DEC-LIST> <COMPOUND-TERM>
  8                    / START <COMPOUND-COMPOOL> <NAME> ;
                         <COMPOUND-TERM>

 *9 <COMPOUND-COMPOOL> ::= COMPOOL
*10                      / <DIRECTIVE> <COMPOUND-COMPOOL>

 11 <PROC-MODULE> ::= START <DEC-LIST>
                      <NON-NESTED-SUB-LIST> <COMPOUND-TERM>
 12                 / START <NON-NESTED-SUB-LIST>
                      <COMPOUND-TERM>

*13 <NON-NESTED-SUB-LIST> ::= <NON-NESTED-SUB>
*14                         / <NON-NESTED-SUB-LIST>
                              <NON-NESTED-SUB>

 15 <NON-NESTED-SUB> ::= DEF <SUB-DEF>
*16                    / <SUB-DEF>

*17 <MAIN-PROGRAM-MODULE>
      ::= START <COMPOUND-PROGRAM> <NAME> ; <COMPOUND-BODY>
          <NON-NESTED-SUB-LIST> <COMPOUND-TERM>

*18     / START <COMPOUND-PROGRAM> <NAME> ; <COMPOUND-BODY>
          <COMPOUND-TERM>

*19     / START <COMPOUND-PROGRAM> <NAME> ; <STMT>
          <NON-NESTED-SUB-LIST> <COMPOUND-TERM>

*20     / START <COMPOUND-PROGRAM> <NAME> ; <STMT>
          <COMPOUND-TERM>
```

```
*21 <COMPOUND-PROGRAM> ::= PROGRAM
*22                      / <DIRECTIVE> <COMPOUND-PROGRAM>

*23 <COMPOUND-BODY> ::= <PROGRAM-BODY>
*24                   / <DIRECTIVE> <COMPOUND-BODY>

*25 <COMPOUND-TERM> ::= TERM
*26                   / <DIRECTIVE> <COMPOUND-TERM>

*27 <PROGRAM-BODY>
      ::= BEGIN <DEC-LIST> <STMT-LIST> <SUB-DEF-LIST>
          <DIRECT-COMPOUND-END>
*28     / BEGIN <DEC-LIST> <STMT-LIST><DIRECT-COMPOUND-EN
*29     / BEGIN <STMT-LIST> <SUB-DEF-LIST>
          <DIRECT-COMPOUND-END>

*30 <DIRECT-COMPOUND-END> ::= <COMPOUND-END>
*31                         / <DIRECTIVE>
                              <DIRECT-COMPOUND-END>

 32 <INTEGER-MACHINE-PARAMETER> ::= BITSINBYTE
 33                               / BITSINWORD
 34                               / LOCSINWORD
 35                               / BYTESINWORD
 36                               / BITSINPOINTER
 37                               / FLOATPRECISION
 38                               / FIXEDPRECISION
 39                               / FLOATRADIX
 40                               / IMPLFLOATPRECISION
                                    ( <FORMULA> )
 41                               / IMPLFIXEDPRECISION (
                                    <FORMULA> , <FORMULA> )
 42                               / IMPLINTSIZE
                                    ( <FORMULA> )
 43                               / MAXFLOATPRECISION
 44                               / MAXFIXEDPRECISION
 45                               / MAXINTSIZE
 46                               / MAXBYTES
 47                               / MAXBITS
 48                               / MAXINT ( <FORMULA> )
 49                               / MININT ( <FORMULA> )
 50                               / MAXSTOP
 51                               / MINSTOP
 52                               / MAXSIGDIGITS
 53                               / MINSIZE ( <FORMULA> )
 54                               / MINFRACTION
                                    ( <FORMULA> )
 55                               / MINSCALE ( <FORMULA> )
 56                               / MINRELPRECISION
                                    ( <FORMULA> )

 57 <FLOATING-MACHINE-PARAMETER>
      ::= MAXFLOAT ( <FORMULA> )
 58     / MINFLOAT ( <FORMULA> )
```

```
 59      / FLOATRELPRECISION ( <FORMULA> )
 60      / FLOATUNDERFLOW ( <FORMULA> )

 61 <FIXED-MACHINE-PARAMETER>
      ::= MAXFIXED ( <FORMULA> , <FORMULA> )
 62     / MINFIXED ( <FORMULA> , <FORMULA> )

*63 <DEC-LIST> ::= <DEC-LIST> <DEC>
*64               / <DEC-LIST> <NULL-STMT> <DEC>
*65               / <NULL-STMT> <DEC>
*66               / <DEC>

*67 <DEC> ::= <DATA-DEC>
*68         / <TYPE-DEC>
*69         / <SUB-DEC>
*70         / <STMT-NAME-DEC>
 71         / <DEFINE-DEC>
 72         / <EXTERNAL-DEC>
 73         / <OVERLAY-DEC>
*74         / <INLINE-DEC>
*75         / BEGIN <DEC-LIST> END
*76         / <DIRECTIVE> <DEC>

 77 <COMPOOL-DEC-LIST> ::= <COMPOOL-DEC-LIST> <COMPOOL-DEC>
 78                       / <COMPOOL-DEC>

 79 <COMPOOL-DEC> ::= <EXTERNAL-DEC>
 80                 / <CONSTANT-DEC>
 81                 / <TYPE-DEC>
 82                 / <DEFINE-DEC>
 83                 / <OVERLAY-DEC>
 84                 / <NULL-STMT>
 85                 / BEGIN <COMPOOL-DEC-LIST> END
 86                 / <DIRECTIVE> <COMPOOL-DEC>

*87 <DATA-DEC> ::= <ITEM-DEC>
*88               / <TABLE-DEC>
*89               / <CONSTANT-DEC>
*90               / <BLOCK-DEC>

*91 <ITEM-DEC> ::= ITEM <NAME> <ALLOCATION-SPECIFIER>
                   <ITEM-TYPE-DESCRIPTION>
                   <ITEM-PRESET-OPTION> ;

*92 <ITEM-TYPE-DESCRIPTION> ::= <INTEGER-ITEM-DESCRIPTION>
*93                             / <FLOATING-ITEM-DESCRIPTION>
*94                             / <FIXED-ITEM-DESCRIPTION>
*95                             / <BIT-ITEM-DESCRIPTION>
*96                             / <CHARACTER-ITEM-
                                        DESCRIPTION>
*97                             / <STATUS-ITEM-DESCRIPTION>
*98                             / <POINTER-ITEM-DESCRIPTION>
*99                             / <ITEM-TYPE-NAME>
```

```
*100 <INTEGER-ITEM-DESCRIPTION> ::= <S-OR-U>
                                    <ROUND-OR-TRUNCATE>
                                      <FORMULA>
*101                              / <S-OR-U> <FORMULA>
*102                              / <S-OR-U>
*103                              / <S-OR-U>
                                    <ROUND-OR-TRUNCATE>

*104 <S-OR-U> ::= S
*105            / U

 106 <FLOATING-ITEM-DESCRIPTION> ::= F <ROUND-OR-TRUNCATE>
                                      <FORMULA>
 107                              / F <FORMULA>
*108                              / F
*109                              / F <ROUND-OR-TRUNCATE>

*110 <ROUND-OR-TRUNCATE> ::= , R
*111                       / , T

 112 <FIXED-ITEM-DESCRIPTION> ::= A <ROUND-OR-TRUNCATE>
                                    <FORMULA> , <FORMULA>
 113                            / A <FORMULA> , <FORMULA>
 114                            / A <FORMULA>
 115                            / A <ROUND-OR-TRUNCATE>
                                    <FORMULA>

 116 <BIT-ITEM-DESCRIPTION> ::= B <FORMULA>
*117                          / B

*118 <CHARACTER-ITEM-DESCRIPTION> ::= C <FORMULA>
*119                                / C

 120 <STATUS-ITEM-DESCRIPTION> ::= STATUS <INTEGER-LITERAL>
                                    ( <STATUS-LIST> )
 121                            / STATUS ( <STATUS-LIST> )

 122 <STATUS-LIST> ::= <STATUS-CONSTANT>
 123                 / <FORMULA> <STATUS-CONSTANT>
 124                 / <STATUS-LIST> , <STATUS-CONSTANT>
 125                 / <STATUS-LIST> , <FORMULA>
                       <STATUS-CONSTANT>

 126 <STATUS-CONSTANT> ::= V ( <STATUS> )

 127 <STATUS> ::= <NAME>
 128            / <LIR>
 129            / <RESERVED-WORD>

 130 <POINTER-ITEM-DESCRIPTION> ::= P <TYPE-NAME>
 131                             / P

 132 <TABLE-DEC> ::= TABLE <NAME> <ALLOCATION-SPECIFICATION>
                     <DIMENSION-LIST> <TABLE-DESCRIPTION>
```

```
133 <TABLE-DESCRIPTION> ::= <STRUCTURE-SPECIFIER>
                             <ENTRY-SPECIFIER>
134                        / <ENTRY-SPECIFIER>
135                        / <TABLE-TYPE-NAME> ;
136                        / <TABLE-TYPE-NAME>
                             <TABLE-PRESET> ;

137 <ENTRY-SPECIFIER> ::= <ORDINARY-ENTRY-SPECIFIER>
138                      / <SPECIFIED-ENTRY-SPECIFIER>

139 <DIMENSION-LIST> ::=
140                    / <DIMENSION-LIST-HEAD> )

141 <DIMENSION-LIST-HEAD> ::= ( <DIMENSION>
142                          / <DIMENSION-LIST-HEAD> ,
                               <DIMENSION>

143 <DIMENSION> ::= <LOWER-BOUND> <FORMULA>
144               / <FORMULA>
145               / *

146 <LOWER-BOUND> ::= <FORMULA> :

147 <STRUCTURE-SPECIFIER> ::= PARALLEL
148                          / T <INTEGER-LITERAL>
149                          / T <FUNCTION-CALL>
150                          / T <NAME>
151                          / T

152 <ORDINARY-ENTRY-SPECIFIER>
      ::= <PACKING-SPECIFIER> <ITEM-TYPE-DESCRIPTION>
          <TABLE-PRESET> ;
153     / <PACKING-SPECIFIER> <ITEM-TYPE-DESCRIPTION> ;
154     / <ITEM-TYPE-DESCRIPTION> <TABLE-PRESET> ;
155     / <ITEM-TYPE-DESCRIPTION> ;
156     / <TABLE-PRESET> ; <ORDINARY-TABLE-BODY>
157     / ; <ORDINARY-TABLE-BODY>
158     / <PACKING-SPECIFIER> <TABLE-PRESET> ;
          <ORDINARY-TABLE-BODY>
159     / <PACKING-SPECIFIER> ; <ORDINARY-TABLE-BODY>

160 <PACKING-SPECIFIER> ::= N
161                       / M
162                       / D

163 <ORDINARY-TABLE-BODY> ::= <ORDINARY-TABLE-ITEM-DEC>
164                         / BEGIN <ORDINARY-TABLE-
                                     OPTIONS-LIST> END

165 <ORDINARY-TABLE-ITEM-DEC>
      ::= ITEM <NAME> <ITEM-TYPE-DESCRIPTION>
          <PACKING-SPECIFIER> <TABLE-PRESET> ;
166     / ITEM <NAME> <ITEM-TYPE-DESCRIPTION>
          <PACKING-SPECIFIER> ;
```

```
167      / ITEM <NAME> <ITEM-TYPE-DESCRIPTION>
           <TABLE-PRESET> ;
168      / ITEM <NAME> <ITEM-TYPE-DESCRIPTION> ;

169 <ORDINARY-TABLE-OPTIONS-LIST>
      ::= <ORDINARY-TABLE-OPTIONS-LIST>
          <ORDINARY-TABLE-OPTIONS>
170      / <ORDINARY-TABLE-OPTIONS>

171 <ORDINARY-TABLE-OPTIONS> ::= <ORDINARY-TABLE-ITEM-DEC>
172                            / <DIRECTIVE>
173                            / <NULL-STMT>

174 <SPECIFIED-ENTRY-SPECIFIER>
      ::= <WORDS-PER-ENTRY> <SPECIFIED-ITEM-DESCRIPTION>
          <TABLE-PRESET> ;
175      / <WORDS-PER-ENTRY> <SPECIFIED-ITEM-DESCRIPTION> ;
176      / <WORDS-PER-ENTRY> <TABLE-PRESET> ;
           <SPECIFIED-TABLE-BODY>
177      / <WORDS-PER-ENTRY> ; <SPECIFIED-TABLE-BODY>

178 <WORDS-PER-ENTRY> ::= W <INTEGER-LITERAL>
179                     / W <FUNCTION-CALL>
180                     / W <NAME>
181                     / *
182                     / V

183 <SPECIFIED-ITEM-DESCRIPTION>
      ::= <ITEM-TYPE-DESCRIPTION> POS
          ( <LOCATION-SPECIFIER> )

184 <LOCATION-SPECIFIER> ::= <STARTING-BIT> , <FORMULA>

185 <STARTING-BIT> ::= <FORMULA>
186                  / *

187 <SPECIFIED-TABLE-BODY> ::= <SPECIFIED-TABLE-ITEM-DEC>
188                          / BEGIN <SPECIFIED-TABLE-
                                  OPTIONS-LIST> END

189 <SPECIFIED-TABLE-ITEM-DEC>
      ::= ITEM <NAME> <SPECIFIED-ITEM-DESCRIPTION>
          <TABLE-PRESET> ;
190      / ITEM <NAME> <SPECIFIED-ITEM-DESCRIPTION> ;

191 <SPECIFIED-TABLE-OPTIONS-LIST>
      ::= <SPECIFIED-TABLE-OPTIONS-LIST>
          <SPECIFIED-TABLE-OPTIONS>
192      / <SPECIFIED-TABLE-OPTIONS>

193 <SPECIFIED-TABLE-OPTIONS> ::= <SPECIFIED-TABLE-ITEM-
                                    DEC>
194                             / <DIRECTIVE>
195                             / <NULL-STMT>
```

```
*196 <CONSTANT-DEC>
      ::= CONSTANT ITEM <NAME> <ITEM-TYPE-DESCRIPTION> =
          <FORMULA> ;
 197    / CONSTANT TABLE <NAME> <DIMENSION-LIST>
          <TABLE-DESCRIPTION>

 198 <BLOCK-DEC>
      ::= BLOCK <NAME> <ALLOCATION-SPECIFIER> ;
          <BLOCK-BODY-PART>
 199    / BLOCK <NAME> <ALLOCATION-SPECIFIER>
          <BLOCK-TYPE-NAME> <BLOCK-PRESET> ;

 200 <BLOCK-BODY-PART> ::= <NULL-STMT>
 201                     / <DATA-DEC>
 202                     / BEGIN <BLOCK-BODY-OPTIONS-LIST>
                           END

 203 <BLOCK-BODY-OPTIONS-LIST> ::= <BLOCK-BODY-OPTIONS-LIST>
                                   <BLOCK-BODY-OPTIONS>
 204                             / <BLOCK-BODY-OPTIONS>

 205 <BLOCK-BODY-OPTIONS> ::= <DATA-DEC>
 206                        / <OVERLAY-DEC>
 207                        / <DIRECTIVE>
 208                        / <NULL-STMT>

*209 <ALLOCATION-SPECIFIER> ::=
 210                          / STATIC

*211 <ITEM-PRESET-OPTION> ::=
 212                        / = <FORMULA>

 213 <TABLE-PRESET> ::= = <TABLE-PRESET-LIST>
 214                  / = <NULL-PRESET>

 215 <NULL-PRESET> ::=

 216 <TABLE-PRESET-LIST> ::= <DEFAULT-PRESET-SUBLIST>
 217                       / <COMPOUND-PRESET-SUBLIST>

 218 <DEFAULT-PRESET-SUBLIST>
      ::= <FORMULA>
 219    / <REPEATED-PRESET-VALUES-OPTION>
 220    / <DEFAULT-PRESET-SUBLIST-HEAD> <FORMULA>
 221    / <DEFAULT-PRESET-SUBLIST-HEAD> <NULL-PRESET>
 222    / <DEFAULT-PRESET-SUBLIST-HEAD>
          <REPEATED-PRESET-VALUES-OPTION>

 223 <DEFAULT-PRESET-SUBLIST-HEAD>
      ::= <PRESET-VALUES-OPTION> ,
 224    / <DEFAULT-PRESET-SUBLIST-HEAD>
          <PRESET-VALUES-OPTION> ,
```

```
225 <COMPOUND-PRESET-SUBLIST>
      ::= <COMPOUND-PRESET-SUBLIST-HEAD> <FORMULA>
226     / <COMPOUND-PRESET-SUBLIST-HEAD> <NULL-PRESET>
227     / <COMPOUND-PRESET-SUBLIST-HEAD>
          <REPEATED-PRESET-VALUES-OPTION>

228 <COMPOUND-PRESET-SUBLIST-HEAD>
      ::= <DEFAULT-PRESET-SUBLIST-HEAD>
          <PRESET-INDEX-SPECIFIER>
229     / <PRESET-INDEX-SPECIFIER>
230     / <COMPOUND-PRESET-SUBLIST-HEAD>
          <PRESET-VALUES-OPTION> ,
231     / <COMPOUND-PRESET-SUBLIST-HEAD>
          <PRESET-INDEX-SPECIFIER>

232 <PRESET-INDEX-SPECIFIER> ::= <PRESET-INDEX-SPECIFIER-
                                 HEAD> ) ;

233 <PRESET-INDEX-SPECIFIER-HEAD>
      ::= POS ( <FORMULA>
234     / <PRESET-INDEX-SPECIFIER-HEAD> ,
                                        <FORMULA>

235 <PRESET-VALUES-OPTION> ::=
236                          <FORMULA>
237                        | <REPEATED-PRESET-VALUES-
                             OPTION>

238 <REPEATED-PRESET-VALUES-OPTION>
      ::= <REPETITION-LIST-HEAD> <FORMULA> )
239     / <REPETITION-LIST-HEAD> <NULL-PRESET> )
240     / <REPETITION-LIST-HEAD>
          <REPEATED-PRESET-VALUES-OPTION> )

241 <REPETITION-LIST-HEAD>
      ::= <INTEGER-LITERAL> (
242     / <REPETITION-LIST-HEAD> <PRESET-VALUES-OPTION> ,

243 <BLOCK-PRESET> ::=
244                  / = <BLOCK-PRESET-LIST>

*245 <TYPE-NAME> ::= <ITEM-TYPE-NAME>
*246                / <TABLE-TYPE-NAME>
*247                / <BLOCK-TYPE-NAME>

*248 <TYPE-DEC> ::= <ITEM-TYPE-DEC>
*249               / <TABLE-TYPE-DEC>
*250               / <BLOCK-TYPE-DEC>

*251 <ITEM-TYPE-DEC> ::= TYPE <NAME><ITEM-TYPE-DESCRIPTION>,

 252 <TABLE-TYPE-DEC> ::= TYPE <NAME> TABLE <TABLE-TYPE-
                          SPECIFIER>
```

```
253 <TABLE-TYPE-SPECIFIER>
      ::= <DIMENSION-LIST> <STRUCTURE-SPECIFIER>
          <LIKE-OPTION> <ENTRY-SPECIFIER>
254     / <DIMENSION-LIST> <STRUCTURE-SPECIFIER>
          <ENTRY-SPECIFIER>
255     / <DIMENSION-LIST> <LIKE-OPTION> <ENTRY-SPECIFIER>
256     / <DIMENSION-LIST> <ENTRY-SPECIFIER>
257     / <DIMENSION-LIST> <TABLE-TYPE-NAME> ;

258 <LIKE-OPTION> ::= LIKE <TABLE-TYPE-NAME>

259 <BLOCK-TYPE-DEC> ::= TYPE <NAME> BLOCK <BLOCK-BODY-
                        PART>

260 <STMT-NAME-DEC> ::= <STMT-NAME-DEC-HEAD> ;

261 <STMT-NAME-DEC-HEAD> ::= LABEL <NAME>
262                       / <STMT-NAME-DEC-HEAD> , <NAME>

263 <DEFINE-DEC> ::= DEFINE <NAME> <DEF-PART>

264 <DEF-PART> ::= <FORMAL-DEFINE-PARAMETER-LIST> <LIST-
                  OPTION> <CHARACTER-STRING> ,

265 <FORMAL-DEFINE-PARAMETER-LIST>
      ::=
266     / <FORMAL-DEFINE-PARAMETER-LIST-HEAD> )

267 <FORMAL-DEFINE-PARAMETER-LIST-HEAD>
      ::= ( <LTR>
268     / <FORMAL-DEFINE-PARAMETER-LIST-HEAD> , <LTR>

269 <LIST-OPTION> ::=
270                 / LISTEXP
271                 / LISTNV
272                 / LISTBOTH

273 <EXTERNAL-DEC> ::= <DEF-SPEC>
274                   / <REF-SPEC>

275 <DEF-SPEC> ::= <SIMPLE-DEF>
276             / <COMPOUND-DEF>

277 <SIMPLE-DEF> ::= DEF <DEF-SPEC-CHOICE>

278 <COMPOUND-DEF> ::= DEF BEGIN <DEF-SPEC-CHOICE-LIST> END

279 <DEF-SPEC-CHOICE-LIST> ::= <DEF-SPEC-CHOICE-LIST>
                               <DEF-SPEC-CHOICE>
280                          / <DEF-SPEC-CHOICE>

281 <DEF-SPEC-CHOICE> ::= <NULL-STMT>
282                     / <DATA-DEC>
283                     / <DEF-BLOCK-INSTANTIATION>
```

```
284                       / <STMT-NAME-DEC>
285                       / <DIRECTIVE> <DEF-SPEC-CHOICE>

286 <DEF-BLOCK-INSTANTIATION> ::= BLOCK INSTANCE <NAME> ;

287 <REF-SPEC> ::= <SIMPLE-REF>
288               / <COMPOUND-REF>

289 <SIMPLE-REF> ::= REF <REF-SPEC-CHOICE>

290 <COMPOUND-REF> ::= REF BEGIN <REF-SPEC-CHOICE-LIST> END

291 <REF-SPEC-CHOICE-LIST> ::= <REF-SPEC-CHOICE-LIST>
                                <REF-SPEC-CHOICE>
292                           / <REF-SPEC-CHOICE>

293 <REF-SPEC-CHOICE> ::= <NULL-STMT>
294                       / <DATA-DEC>
295                       / <SUB-DEC>
296                       / <DIRECTIVE> <REF-SPEC-CHOICE>

297 <OVERLAY-DEC> ::= OVERLAY <ABSOLUTE-ADDRESS>
                      <OVERLAY-EXPRESSION> ;

298 <ABSOLUTE-ADDRESS> ::=
299                        / POS ( <FORMULA> ) :

300 <OVERLAY-EXPRESSION>
      ::= <OVERLAY-STRING>
301     / <OVERLAY-EXPRESSION> : <OVERLAY-STRING>

302 <OVERLAY-STRING>
      ::= <OVERLAY-ELEMENT>
303     / <OVERLAY-STRING> , <OVERLAY-ELEMENT>

304 <OVERLAY-ELEMENT> ::= <SPACER>
305                       / <NAME>
306                       / ( <OVERLAY-EXPRESSION> )

307 <SPACER> ::= W <FORMULA>

*308 <SUB-DEC> ::= <PROC-DEC>
*309              / <FUNCTION-DEC>

*310 <SUB-DEF-LIST> ::= <SUB-DEF>
*311                   / <SUB-DEF-LIST> <SUB-DEF>

*312 <SUB-DEF> ::= <PROC-DEF>
*313              / <FUNCTION-DEF>
*314              / <DIRECTIVE> <SUB-DEF>

*315 <PROC-DEC> ::= <PROC-HEADING> ; <DEC>

*316 <PROC-DEF> ::= <PROC-HEADING> ; <COMPOUND-BODY>
```

```
*317                / <PROC-HEADING> ; <STMT>

*318 <PROC-HEADING> ::= PROC <NAME> <SUB-ATTRIBUTE>
                        <FORMAL-PARAMETER-LIST>

*319 <SUB-ATTRIBUTE> ::=
*320                    / REC
*321                    / RENT

*322 <FUNCTION-DEC> ::= <FUNCTION-HEADING> ; <DEC>

*323 <FUNCTION-DEF> ::= <FUNCTION-HEADING> ; <COMPOUND-BODY>
*324                  / <FUNCTION-HEADING> ; <STMT>

*325 <FUNCTION-HEADING> ::= PROC <NAME> <SUB-ATTRIBUTE>
                            <FORMAL-PARAMETER-LIST>
                            <ITEM-TYPE-DESCRIPTION>

*326 <FORMAL-PARAMETER-LIST>
       ::=
 327     / ( : <FORMAL-IO-PARAMETER-LIST> )
 328     / ( <FORMAL-IO-PARAMETER-LIST> :
           <FORMAL-IO-PARAMETER-LIST> )
 329     / ( <FORMAL-IO-PARAMETER-LIST> )

 330 <FORMAL-IO-PARAMETER-LIST>
       ::= <FORMAL-IO-PARAMETER-LIST> , <NAME>
 331     / <NAME>

*332 <INLINE-DEC> ::= <INLINE-DEC-HEAD> ;

*333 <INLINE-DEC-HEAD> ::= INLINE <SUB-NAME>
*334                     / <INLINE-DEC-HEAD> , <SUB-NAME>

*335 <SUB-NAME> ::= <PROC-LABEL-NAME>
*336              / <NAME>

*337 <STMT-LIST> ::= <STMT-LIST> <STMT>
*338               / <STMT-LIST> <NULL-STMT> <STMT>
*339               / <NULL-STMT> <STMT>
*340               / <STMT>

*341 <STMT> ::= <BALANCED-STMT>
*342          / <UNBALANCED-STMT>

*343 <BALANCED-STMT> ::= <BALANCED>
*344                   / <DIRECTIVE> <BALANCED-STMT>

*345 <UNBALANCED-STMT> ::= <UNBALANCED>
*346                     / <DIRECTIVE> <UNBALANCED-STMT>

*347 <UNBALANCED> ::= <LABEL> <UNBALANCED>
*348                / <UNBALANCED-IF-STMT>
*349                / <UNBALANCED-FOR-STMT>
```

```
*350                  / <UNBALANCED-WHILE-STMT>

*351 <BALANCED> ::= <LABEL> <BALANCED>
*352                / <ASSIGNMENT-STMT> ;
*353                / <BALANCED-FOR-STMT>
*354                / <BALANCED-WHILE-STMT>
*355                / <BALANCED-IF-STMT>
*356                / <CASE-STMT>
*357                / <PROC-CALL-STMT>
*358                / <RETURN-STMT> ;
*359                / <GOTO-STMT> ;
*360                / <EXIT-STMT> ;
 361                / <STOP-STMT> ;
 362                / <ABORT-STMT> ;
*363                / BEGIN <STMT-LIST> <DIRECT-COMPOUND-END>

*364 <NULL-STMT> ::= ;
*365                / BEGIN <COMPOUND-END>

*366 <COMPOUND-END> ::= END
*367                  / <LABEL> <COMPOUND-END>

*368 <LABEL> ::= <PROC-LABEL-NAME> :

*369 <ASSIGNMENT-STMT> ::= <TARGET-LIST> = <EXPRESSION>

 370 <TARGET-LIST> ::= <TARGET-LIST> , <TARGET>
*371                  / <TARGET>

*372 <TARGET> ::= <LHS>
*373             / <PROC-LABEL-NAME>

 374 <BALANCED-FOR-STMT> ::= <FOR-CLAUSE> <BALANCED-STMT>
 375                       / <FOR-BY> <BALANCED-STMT>
 376                       / <FOR-THEN> <BALANCED-STMT>

 377 <UNBALANCED-FOR-STMT>
       ::= <FOR-CLAUSE> <UNBALANCED-STMT>
 378     / <FOR-BY> <UNBALANCED-STMT>
 379     / <FOR-THEN> <UNBALANCED-STMT>

 380 <FOR-CLAUSE> ::= <INITIAL> <SAFE>
 381                / <INITIAL> <WHILE-PHRASE> <SAFE>

 382 <FOR-BY>
       ::= <INITIAL> <BY-PHRASE> <SAFE>
 383     / <INITIAL> <BY-PHRASE> <WHILE-PHRASE> <SAFE>
 384     / <INITIAL> <WHILE-PHRASE> <BY-PHRASE> <SAFE>

 385 <FOR-THEN>
       ::= <INITIAL> <THEN-PHRASE> <SAFE>
 386     / <INITIAL> <THEN-PHRASE> <WHILE-PHRASE> <SAFE>
 387     / <INITIAL> <WHILE-PHRASE> <THEN-PHRASE> <SAFE>
```

```
 388 <INITIAL> ::= FOR <NAME> : <EXPRESSION>
 389             / FOR <LTR> : <EXPRESSION>

 390 <BY-PHRASE> ::= BY <EXPRESSION>

 391 <THEN-PHRASE> ::= THEN <EXPRESSION>

 392 <WHILE-PHRASE> ::= WHILE <BRANCH-FALSE>

*393 <BRANCH-FALSE> ::= <BIT-FORMULA>

*394 <BALANCED-WHILE-STMT> ::= <WHILE-CLAUSE>
                                <BALANCED-STMT>

*395 <UNBALANCED-WHILE-STMT> ::= <WHILE-CLAUSE>
                                   <UNBALANCED-STMT>

*396 <WHILE-CLAUSE> ::= WHILE <BRANCH-FALSE> <SAFE>

*397 <SAFE> ::= ;
 398          / ; !SAFE ;

*399 <BALANCED-IF-STMT> ::= <IF-PREFIX> <BALANCED-STMT>

*400 <UNBALANCED-IF-STMT> ::= <IF-CLAUSE> <STMT>
*401                        / <IF-PREFIX> <UNBALANCED-STMT>

 402 <IF-PREFIX> ::= <IF-CLAUSE> <BALANCED-STMT> ELSE

*403 <IF-CLAUSE> ::= IF <BRANCH-FALSE> ;

*404 <CASE-STMT> ::= <CASE-BODY> <COMPOUND-END>

*405 <CASE-BODY> ::= <CASE-CLAUSE> <CASE-CHOICE>
*406               / <CASE-BODY> <CASE-CHOICE>

*407 <CASE-CLAUSE> ::= CASE <EXPRESSION> ; BEGIN

*408 <CASE-CHOICE> ::= <CASE-ALT> : <STMT>
 409                 / <CASE-ALT> : <STMT> FALLTHRU

*410 <CASE-ALT> ::= ( <CASE-INDEX-GROUP> )
*411              / ( DEFAULT )

*412 <CASE-INDEX-GROUP>
        ::= <CASE-INDEX>
*413      / <CASE-INDEX-GROUP> , <CASE-INDEX>

*414 <CASE-INDEX> ::= <EXPRESSION>
*415                / <EXPRESSION> : <EXPRESSION>

 416 <PROC-CALL-STMT>
        ::= <INVOCATION> ABORT <PROC-LABEL-NAME> ;
*417      / <INVOCATION> ;
```

```
*418 <INVOCATION> ::= <OUTPUT-LIST> )
*419                / <INPUT-LIST> )
*420                / <PROC-LABEL-NAME>

*421 <INPUT-LIST> ::= <CALL-PREFIX> <INPUT-PARM>
*422                / <INPUT-LIST> , <INPUT-PARM>

*423 <OUTPUT-LIST> ::= <INPUT-LIST> : <OUTPUT-PARM>
*424                 / <CALL-PREFIX> : <OUTPUT-PARM>
*425                 / <OUTPUT-LIST> , <OUTPUT-PARM>

*426 <CALL-PREFIX> ::= <PROC-LABEL-NAME> (

*427 <INPUT-PARM> ::= <EXPRESSION>

*428 <OUTPUT-PARM> ::= <LHS>

*429 <RETURN-STMT> ::= RETURN

*430 <GOTO-STMT> ::= GOTO <PROC-LABEL-NAME>

*431 <EXIT-STMT> ::= EXIT

 432 <STOP-STMT> ::= STOP
 433               / STOP <FORMULA>

 434 <ABORT-STMT> ::= ABORT

*435 <FORMULA> ::= <FORMULA> + <TERM>
*436             / <FORMULA> - <TERM>
*437             / + <TERM>
*438             / - <TERM>
*439             / <TERM>

*440 <TERM> ::= <TERM> * <FACTOR>
*441          / <TERM> / <FACTOR>
*442          / <TERM> MOD <FACTOR>
*443          / <FACTOR>

*444 <FACTOR> ::= <FACTOR> ** <PRIMARY>
*445             / <PRIMARY>

*446 <PRIMARY> ::= <INTEGER-LITERAL>
*447              / <REAL-LITERAL>
*448              / <BIT-LITERAL>
*449              / <CHARACTER-LITERAL>
*450              / <BOOLEAN-LITERAL>
*451              / <POINTER-LITERAL>
*452              / <NAMED-VARIABLE>
*453              / <LETTER>
*454              / <FUNCTION-CALL>
 455              / <INTEGER-MACHINE-PARAMETER>
 456              / <FLOATING-MACHINE-PARAMETER>
```

```
 457              / <FIXED-MACHINE-PARAMETER>
*458              / ( <EXPRESSION> )
 459              / <CONVERSION> ( <EXPRESSION> )

*460 <BIT-FORMULA> ::= <AND-FORMULA>
*461                 / <OR-FORMULA>
*462                 / <XOR-FORMULA>
*463                 / <EQV-FORMULA>
*464                 / <BIT-PRIMARY>
*465                 / NOT <BIT-PRIMARY>

*466 <BIT-PRIMARY> ::= <RELATIONAL-EXPRESSION>
*467                 / <FORMULA>

*468 <AND-FORMULA> ::= <BIT-PRIMARY> AND <BIT-PRIMARY>
*469                 / <AND-FORMULA> AND <BIT-PRIMARY>

*470 <OR-FORMULA> ::= <BIT-PRIMARY> OR <BIT-PRIMARY>
*471                / <OR-FORMULA> OR <BIT-PRIMARY>

*472 <XOR-FORMULA> ::= <BIT-PRIMARY> XOR <BIT-PRIMARY>
*473                 / <XOR-FORMULA> XOR <BIT-PRIMARY>

 474 <EQV-FORMULA> ::= <BIT-PRIMARY> EQV <BIT-PRIMARY>
 475                 / <EQV-FORMULA> EQV <BIT-PRIMARY>

*476 <RELATIONAL-EXPRESSION>
       ::= <FORMULA> <RELATIONAL-OPERATOR> <FORMULA>

*477 <EXPRESSION> ::= <BIT-FORMULA>

*478 <NAMED-VARIABLE> ::= <SUBSCRIPT> )
*479                    / <NAME>
 480                    / <SUBSCRIPT> )  <POINTER>
 481                    / <NAME>  <POINTER>
 482                    /  <POINTER>

*483 <LTR> ::= <LETTER>
*484         / A
*485         / B
*486         / C
*487         / D
*488         / F
*489         / M
*490         / N
*491         / P
*492         / R
*493         / S
*494         / T
*495         / U
*496         / V
*497         / W

 498 <POINTER> ::= <NAME>
```

```
 499                 / ( <FORMULA> )

*500 <SUBSCRIPT> ::= <PREFIX> <FORMULA>
*501                 / <SUBSCRIPT> , <FORMULA>

*502 <PREFIX> ::= <NAME> (
 503             /  <POINTER> (

*504 <LHS> ::= <NAMED-VARIABLE>
 505          / <PSEUDO-VARIABLE>

 506 <PSEUDO-VARIABLE>
       ::= BIT ( <TARGET> , <FORMULA> , <FORMULA> )
 507     / BYTE ( <TARGET> , <FORMULA> , <FORMULA> )
 508                       / REP ( <NAMED-VARIABLE> )

*509 <FUNCTION-CALL> ::= <INVOCATION>
 510                    / <INTRINSIC-FUNCTION-CALL>

 511 <INTRINSIC-FUNCTION-CALL> ::= <LOC-FUNCTION>
 512                               / <NEXT-FUNCTION>
 513                               / <BIT-FUNCTION>
 514                               / <BYTE-FUNCTION>
 515                               / <SHIFT-FUNCTION>
 516                               / <ABS-FUNCTION>
 517                               / <SIGN-FUNCTION>
 518                               / <SIZE-FUNCTION>
 519                               / <BOUNDS-FUNCTION>
 520                               / <NWDSEN-FUNCTION>
 521                               / <STATUS-INVERSE-FUNCTION>

 522 <LOC-FUNCTION> ::= LOC ( <NAMED-VARIABLE> )

 523 <NEXT-FUNCTION> ::= NEXT ( <FORMULA> , <FORMULA> )

 524 <BIT-FUNCTION>
       ::= BIT ( <EXPRESSION> , <FORMULA> , <FORMULA> )

 525 <BYTE-FUNCTION>
       ::= BYTE ( <FORMULA> , <FORMULA> , <FORMULA> )

 526 <SHIFT-FUNCTION>
       ::= <SHIFT-DIRECTION> ( <EXPRESSION> , <FORMULA> )

 527 <SHIFT-DIRECTION> ::= SHIFTL
 528                     / SHIFTR

 529 <ABS-FUNCTION> ::= ABS ( <FORMULA> )

 530 <SIGN-FUNCTION> ::= SGN ( <FORMULA> )

 531 <SIZE-FUNCTION> ::= <SIZE-TYPE> ( <FORMULA> )
 532                   / <SIZE-TYPE> ( <TYPE-NAME> )
```

```
533 <SIZE-TYPE> ::= BITSIZE
534                 / BYTESIZE
535                 / WORDSIZE

536 <BOUNDS-FUNCTION>
      ::= <WHICH-BOUND> ( <NAME> , <FORMULA> )

537 <WHICH-BOUND> ::= LBOUND
538                  / UBOUND

539 <NWDSEN-FUNCTION> ::= NWDSEN ( <NAME> )
540                      / NWDSEN ( <TABLE-TYPE-NAME> )

541 <STATUS-INVERSE-FUNCTION>
      ::= FIRST ( <STATUS-INVERSE-ARGUMENT> )
542     / LAST ( <STATUS-INVERSE-ARGUMENT> )

543 <STATUS-INVERSE-ARGUMENT> ::= <FORMULA>
544                              / <ITEM-TYPE-NAME>

545 <CONVERSION> ::= (* <ITEM-TYPE-DESCRIPTION> *)
546                 / <TYPE-NAME>
547                 / REP
548                 / B
549                 / C
550                 / F
551                 / P
552                 / S
553                 / U

554 <NAME-LIST> ::= <NAME>
555                / <NAME-LIST> , <NAME>

556 <RESERVED-WORD> ::= ABORT
557                    / ABS
558                    / AND
559                    / BEGIN
560                    / BIT
561                    / BITSIZE
562                    / BLOCK
563                    / BY
564                    / BYTE
565                    / BYTESIZE
566                    / CASE
567                    / COMPOOL
568                    / CONSTANT
569                    / DEF
570                    / DEFAULT
571                    / DEFINE
572                    / ELSE
573                    / END
574                    / EQV
575                    / FALLTHRU
576                    / FALSE
```

```
 577                        / FIRST
 578                        / FOR
 579                        / GOTO
 580                        / IF
 581                        / INLINE
 582                        / INSTANCE
 583                        / ITEM
 584                        / LABEL
 585                        / LAST
 586                        / LBOUND
 587                        / LIKE
 588                        / LOC
 589                        / MOD
 590                        / NEXT
 591                        / NOT
 592                        / NULL
 593                        / NWDSEN
 594                        / OR
 595                        / OVERLAY
 596                        / PARALLEL
 597                        / POS
 598                        / PROC
 599                        / PROGRAM
 600                        / REC
 601                        / REF
 602                        / RENT
 603                        / REP
 604                        / RETURN
 605                        / SGN
 606                        / SHIFTL
 607                        / SHIFTR
 608                        / START
 609                        / STATIC
 610                        / STATUS
 611                        / STOP
 612                        / TABLE
 613                        / TERM
 614                        / THEN
 615                        / TRUE
 616                        / UBOUND
 617                        / WHILE
 618                        / WORDSI
 619                        / XOR

*620 <RELATIONAL-OPERATOR> ::= =
*621                            / <>
*622                            / <
*623                            / >
*624                            / <=
*625                            / >=

 626 <BIT-LITERAL> ::= <INTEGER-LITERAL> B
                       <CHARACTER-LITERAL>
```

```
*627 <BOOLEAN-LITERAL> ::= TRUE
*628                     / FALSE

*629 <POINTER-LITERAL> ::= NULL

 630 <DIRECTIVE> ::= <COMPOOL-DIRECTIVE>
 631                / <COPY-DIRECTIVE>
 632                / <SKIP-DIRECTIVE>
 633                / <BEGIN-DIRECTIVE>
 634                / <END-DIRECTIVE>
 635                / <LINKAGE-DIRECTIVE>
 636                / <TRACE-DIRECTIVE>
 637                / <INTERFERENCE-DIRECTIVE>
 638                / <REDUCIBLE-DIRECTIVE>
 639                / <NOLIST-DIRECTIVE>
 640                / <LIST-DIRECTIVE>
 641                / <EJECT-DIRECTIVE>
 642                / <BASE-DIRECTIVE>
 643                / <ISBASE-DIRECTIVE>
 644                / <DROP-DIRECTIVE>
 645                / <LEFTRIGHT-DIRECTIVE>
 646                / <REARRANGE-DIRECTIVE>
 647                / <INITIALIZE-DIRECTIVE>
 648                / <ORDER-DIRECTIVE>

 649 <COMPOOL-DIRECTIVE>
       ::= !COMPOOL <COMPOOL-DIRECTIVE-LIST> ,

 650 <COMPOOL-DIRECTIVE-LIST>
       ::=
 651     / <COMPOOL-FILE-NAME> <COMPOOL-DECLARED-NAME-LIST>
 652     / <COMPOOL-DECLARED-NAME-LIST>
 653     / ( )
 654     / ( <COMPOOL-FILE-NAME> )

 655 <COMPOOL-DECLARED-NAME-LIST>
       ::= <COMPOOL-DECLARED-NAME>
 656     / <COMPOOL-DECLARED-NAME-LIST> ,
         <COMPOOL-DECLARED-NAME>

 657 <COMPOOL-DECLARED-NAME> ::= <NAME>
 658                          / <TYPE-NAME>
 659                          / <PROC-LABEL-NAME>
 660                          / ( <TYPE-NAME> )
 661                          / ( <PROC-LABEL-NAME> )
 662                          / ( <NAME> )

 663 <COMPOOL-FILE-NAME> ::= <CHARACTER-LITERAL>

 664 <COPY-DIRECTIVE> ::= !COPY <CHARACTER-LITERAL> ;

 665 <SKIP-DIRECTIVE> ::= !SKIP ;
 666                   / !SKIP <LIT> ;
```

```
667 <BEGIN-DIRECTIVE> ::= !BEGIN ;
668                       / !BEGIN <LTR> ;

669 <END-DIRECTIVE> ::= !END ;

670 <LINKAGE-DIRECTIVE> ::= !LINKAGE <SYMBOL-LIST> ;

671 <TRACE-DIRECTIVE>
      ::= !TRACE ( <FORMULA> ) <NAME-LIST> ;
672     / !TRACE <NAME-LIST>

673 <INTERFERENCE-DIRECTIVE>
      ::= !INTERFERENCE <INTERFERENCE-CONTROL> ;

674 <INTERFERENCE-CONTROL>
      ::= <NAME> : <NAME>
675     / <INTERFERENCE-CONTROL> , <NAME>

676 <REDUCIBLE-DIRECTIVE> ::= !REDUCIBLE ;

677 <NOLIST-DIRECTIVE> ::= !NOLIST ;

678 <LIST-DIRECTIVE> ::= !LIST ;

679 <EJECT-DIRECTIVE> ::= !EJECT ;

680 <BASE-DIRECTIVE> ::= !BASE <NAME> <INTEGER-LITERAL> ;

681 <ISBASE-DIRECTIVE> ::= !ISBASE <NAME><INTEGER-LITERAL>,

682 <DROP-DIRECTIVE> ::= !DROP <INTEGER-LITERAL> ;

683 <LEFTRIGHT-DIRECTIVE> ::= !LEFTRIGHT ;

684 <REARRANGE-DIRECTIVE> ::= !REARRANGE ;

685 <INITIALIZE-DIRECTIVE> ::= !INITIALIZE ;

686 <ORDER-DIRECTIVE> ::= !ORDER ;
```

Appendix C

System Maintenance Guide

This appendix describes the structure and organization of JATS. The purpose of this appendix is to provide the necessary information to the maintenance programmer to allow modification of the system. The overall system will be briefly described followed by a system structure chart, a description of each subroutine, and a list of system error statements.

JATS uses the LR parser generator system to construct a parse tree of the input J73 source program. This parse tree is then translated so that it represents an equivalent Ada source program. Finally, the terminal nodes are output using a prettyprint format. The structure of JATS can therefore be grouped into three functional modules: (1) parser module, (2) translation module, and (3) prettyprint module.

The parser module was designed from the LR parser generator system and the parsing algorithms were taken directly from LR. Appendix A contains a brief description of LR and the data structures that support the parsing algorithms. There is one modification to the parsing routines to accommodate the J73 grammar. During the parsing loop, a check is made in certain states of the parser to set a flag which indicates whether the parser is processing

executable statements or declarations. This flag is used within the scanner to return the correct token during scanning of <name>s. The major additions to the parser are the scanner and the semantic processor. The scanner returns the tokens of the input stream to the parser. It consists of a number of subroutines which will be discussed later. The semantic processor constructs the parse tree during the reduction processes. The subroutines that support the semantic processor will be discussed later. One other area of interest is the use of a "heap" for dynamic allocation of the nodes and elements of the tree.

The translation module processes the parse tree using the language production numbers that were used to construct each internal node. Worth noting during the translation process is the reclamation and reallocation of heap space. A list of available heap space is maintained by linking free spaces together. Nodes of the tree that are deleted are also made available for reallocation through a linked list of available nodes using an overlayed data structure.

The prettyprint module simply outputs the leaves, terminal nodes, of the parse tree and uses the symbols to control the formatting of the resulting Ada program.

A system structure chart is spread across figures 12, 13, and 14. A brief description of each subroutine follows the system structure charts. Also included in this appendix, is table IV which describes the possible syste

errors and the changes required to correct the error.

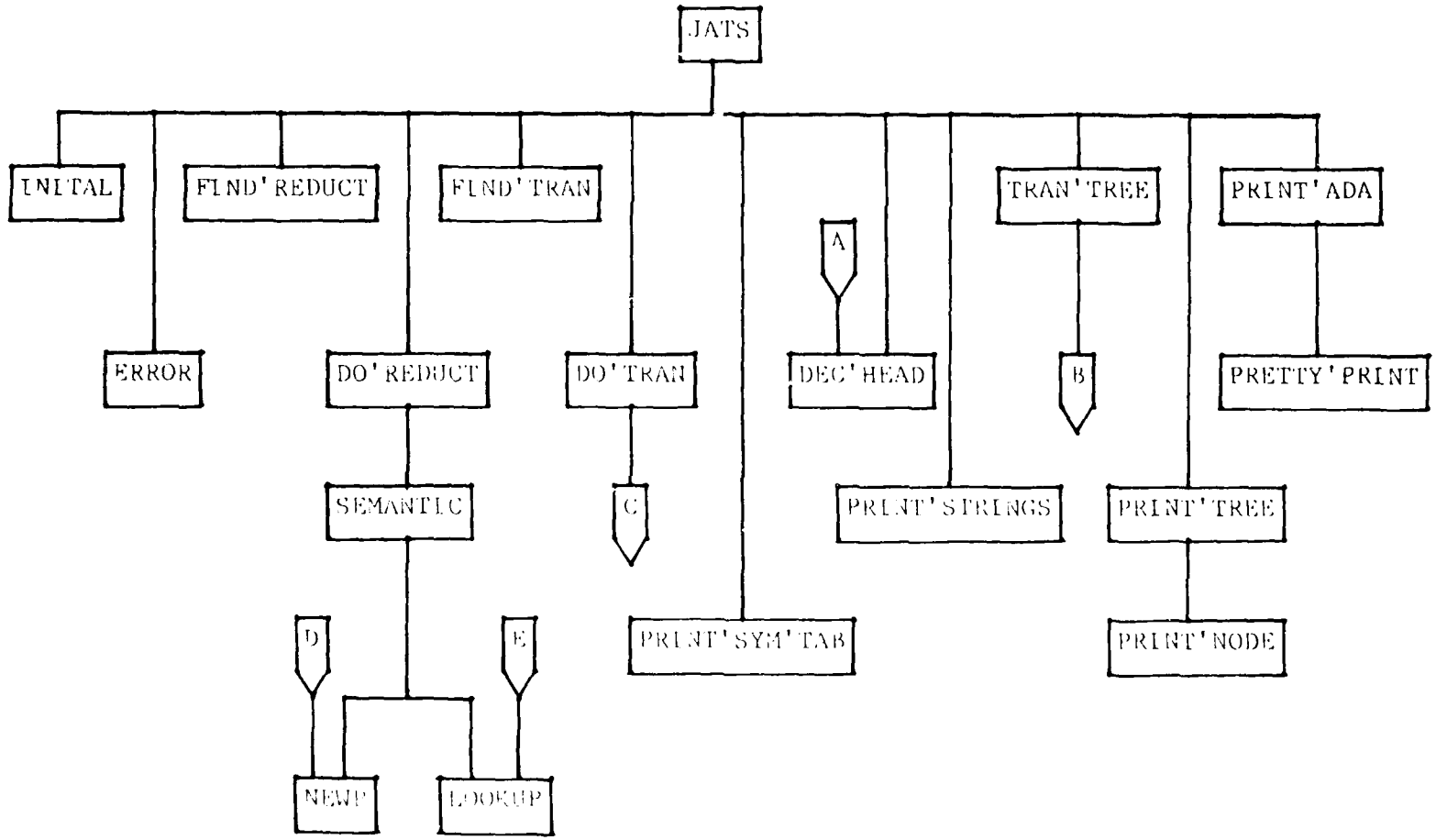


Fig 12. System Structure Chart

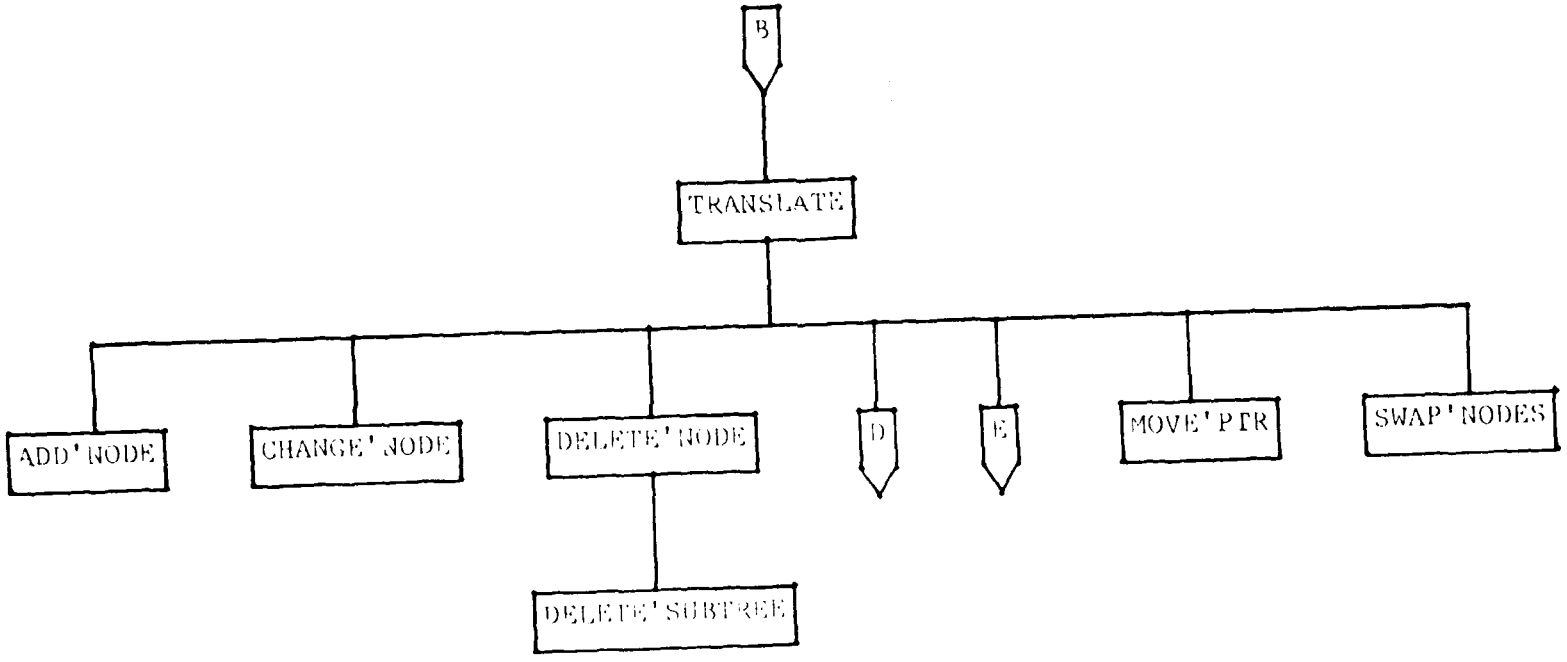


Fig. 13. System Structure Chart

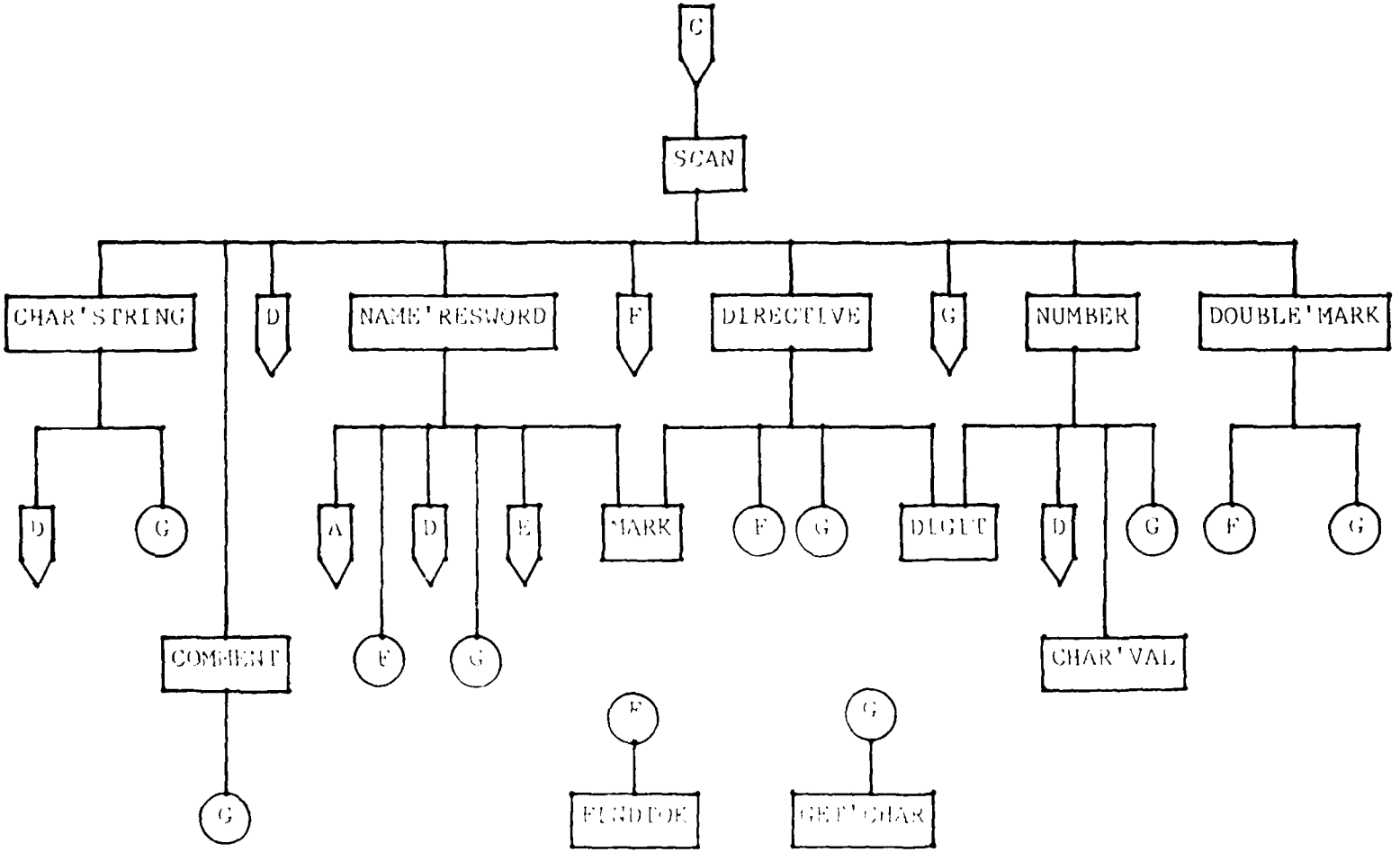


Fig 14. Scanner Structure Chart

ADD'NODE. Local to procedure TRANSLATE. Used during the translation process to create a new terminal node consisting of the terminal symbol TERM'CODE, it adds the new element to the current node. The new element is inserted immediately after the element pointed to by PTR. Updates the global variable NEXT'NODE.

CHANGE'NODE. Local to procedure TRANSLATE. Changes the terminal symbol at the node identified by the element pointed to by PTR to TERM'CODE.

CHAR'STRING. Local to procedure SCAN. Used to scan <character-literal>s during the parsing process. Constructs a new element of the terminal node to identify the <character-literal>. Updates the global variables NEXT'STRING and STRING(NEXT'STRING). Returns the token number of <character-literal> in SCAN'TOK.

CHAR'VAL. Local to procedure NUMBER. Function used during the process of scanning numbers to return the value of the digit CH.

COMMENT. Local to procedure SCAN. Used to scan a comment.

DEC'HEAD. Global boolean function. Used during the parsing process. Returns the value TRUE if the token, IN'TOKEN, is the start of a declaration. If IN'TOKEN is a <name> then the global CUR'TOKEN is changed to a <proc-label-name>.

DELETE'NODE. Local to procedure TRANSLATE. Used during translation to delete the node element immediately following the element identified by PTR and calls DELETE'SUBTREE with the node number at the deleted element. Updates the global variable FREE'PTR.

DELETE'SUBTREE. Local to procedure DELETE'NODE. This is a recursive procedure that deletes all nodes and elements from the parse tree as a result of deleting the node NODE'NUMBER. Updates the global variables NEXT'NODE and FREE'PTR.

DIGIT. Local to procedure SCAN. Boolean function which returns the value TRUE if CH is a digit.

DIRECTIVE. Local to procedure SCAN. Used to scan directives. Returns the proper token number in SCAN'TOP.

DO'REDUCT. Global procedure. Used during the parsing process to perform a reduction based on the length of production PROD'NUM. Calls SEMANTIC and pushes the new information onto the parsing stacks. Updates the global variables STACK'PTR and CUR'STATE.

DO'TRAN. Global procedure. Used during the parsing process to perform a transition to NEW'STATE. Stacks the current information onto the parsing stacks and calls SCAN for a new CUR'TOKEN. Updates the global variables CUR'STATE and CUR'TOKEN.

DOUBLE'MARK. Local to procedure SCAN. Checks if the current token being scanned might be a double character symbol, such as <>. Returns the correct token number as SCAN'TOK for either case.

ERROR. Global procedure. Used during the parsing process to output information regarding the current status of the parser when a syntax error has been detected. Sets the global variable ERR'FLAG to the value TRUE which halts the parser and translator system.

FIND'REDUCT. Global procedure. Function used during the parsing process to find the proper reduction to perform based upon CSTATE and CTOKEN.

FINDTOK. Local to procedure SCAN. Used during the scanning of terminal symbols. Uses TOK'SYMBOL and LENGTH to match the current symbol with the terminal symbols of J73, if a match is found the value of the token number is returned as RET'VALUE.

FIND'TRAN. Global procedure. Function used during the parsing process to find the proper transition based upon CSTATE and CTOKEN.

GET'CHAR. Local to procedure SCAN. Returns the next character of the input stream as CH, and echochecks the input line images. Sets EOF'FLAG and clears NEXT'CHAR.

INITAL. Global procedure. Initializes various global

variables and data structures.

LOOKUP. Global procedure. Uses the global variable SYMBOL to find the symbol table entry for the <name> that is contained in SYMBOL with a length of SYM'LENGTH, a global variable. If an entry is found, ENTRY'PTR contains the pointer value of the entry and FOUND is TRUE. If an entry is not found, the symbol is entered in the symbol table and ENTRY'PTR contains the pointer value of the new entry and FOUND is FALSE. If a new entry is made the global variable NEXT'SYMBOL is updated.

MARK. Local to procedure SCAN. Boolean function to determine if the current character of the input stream is a mark, or seperator. Returns TRUE is the character is a mark.

MOVE'PTR. Local to procedure TRANSLATE. Used during translation to move the element pointer of a node, PTR, the number of positions specified by COUNT.

NAME'RESWORD. Local to procedure SCAN. Used to scan <name>s and reserved words. Once the symbol has been scanned, if FINDTOK returns a value 0 then it is a <name>. Builds the global variable SYMBOL and SYM'LENGTH. Returns the correct token number as SCAN'TOK. May reset the global variable STMT'SCAN'FLAG.

NEWP. Global procedure. Function which returns the next available heap space pointer address for a data

structure of the given SIZE. Updates the global variable FREE'PTR or NEXT'HEAP as necessary.

NUMBER. Local to procedure SCAN. Used to scan numbers and returns the proper token number as SCAN'TOK and constructs a node that contains the value of the appropriate literal.

PRETTY'PRINT. Local to procedure PRINT'ADA. Used to output the terminal symbol contained in the terminal node PRINT'NODE.

PRINT'ADA. Global procedure. Recursive procedure which traverses the subtrees of NODE'NUMBER and calls PRETTY'PRINT when a terminal node is reached. Also outputs warnings that certain sections of code have not been fully translated. Sets and resets the global variable LAST'TRAN'FLAG.

PRINT'NODE. Global procedure. Utility procedure that outputs the contents of the node NODE'NUMBER.

PRINT'STRINGS. Global procedure. Utility procedure that outputs the <character-literal>s that have been parsed.

PRINT'SYM'TAB. Global procedure. Utility procedure that outputs the contents of the symbol table.

PRINT'TREE. Global procedure. Recursive procedure that traverses the parse tree and calls PRINT'NODE to output the entire parse tree.

SCAN. Global procedure. Scans the input and returns the token numbers as SCAN'TOK. Creates a new terminal node each time it is called and updates the global variable NEXT'NODE.

SEMANTIC. Global procedure. Used during the parsing process to construct a new node based on the language production PROD'NUM. Also enters the SYM'CLASS values in the symbol table and updates the global variables ROOT, NEXT'NODE, and NEW'TREE'PTR.

SWAP'NODES. Local to procedure TRANSLATE. Used during the translation process to swap the two node elements that immediately follow the element pointer to by PTR. PTR is moved to point to the new element that follows the element that was initially pointed to by PTR.

TRANSLATE. Local to procedure TRAN'TREE. Controls the translation process at each node TRAN'NODE. Sets the TRAN'FLAG for each node that is translated. May create new nodes and elements.

TRAN'TREE. Global procedure. Recursive procedure used to traverse the parse tree during translation.

Table IV

System Errors

Error Number	Cause/Required Changes
1	Parsing stack overflow Increase the value of MAX'STACK
2	SYM'STORE'TAB overflow Increase the value of MAX'SYM'STORE
3	HEAP overflow Increase the value of MAX'HEAP
4	TREE overflow Increase the value of MAX'NODES
5	STRING overflow Increase the value of MAX'STRINGS
6	STRING'STORE overflow Increase the value of MAX'STRING'STORE

Appendix D

JATS User Guide

The purpose of this appendix is to provide some guidance for JATS users. The system itself is very easy to use; however, there are several limitations that the user must adhere to for JATS to work properly. As shown in table I, there are several elements of J73 that have modified definitions.

One other area that the user must prepare for translation is the identifiers or <name>s of the J73 program. As JATS is currently implemented, <name>s that are to be translated must not be Ada reserved words. The other limitation with respect to <name>s is the use of the single quote within the identifiers. The use of the single quote must conform to the rules of the use of the underscore in Ada. This requires that the single quote be preceeded and followed by either a digit or letter and the single quote cannot end the <name>. Also, the use of the dollar sign is not allowed since it does not translate to any equivalent element of Ada. These limitations can be removed when JATS is modified to become an interactive system to allow the user to rename an identifier when necessary.

With these limitations taken care of, the J73 program can be translated with JATS. On the AFWAL/AA DecSystem-10, JATS can be run with the following command:

```
.RUN JATS[5600,113]
*<file name>.<extension>
```

JATS will respond with an asterisk as a prompt for the file name at which time the user should respond with a valid file name with an extension of J73.

JATS will create two output files: (1) JATS.LST and (2) <file name>.ADA. JATS.LST will contain a listing of the input program and any information messages that were generated. The second file is the translated version of the program and will have the same file name with an extension of ADA. This file will contain warning messages identifying sections of code that were not translated. These sections will be bracketed by the following messages.

```
----------------------------WARNING-----------------------------
--                                                              --
--     THE CODE BETWEEN THIS BRACKET AND THE FOLLOWING          --
--           BRACKET MAY NOT BE FULLY TRANSLATED                --
--                                                              --


--                                                              --
--     THE CODE BETWEEN THIS BRACKET AND THE PREVIOUS           --
--           BRACKET MAY NOT BE FULLY TRANSLATED                --
--                                                              --
----------------------------WARNING-----------------------------
```

JATS has been designed to process large programs but does have some limitations on size. If a message consisting of a JATS system error appears at the terminal while JATS is

running, then one of the size limitations has been exceeded and JATS will have to be modified and recompiled. If this occurs, notify the maintenance programmer.

An example J73 program and its translated version are included in Appendix E.

AD-A100 881 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOO--ETC F/6 9/2
JOVIAL (J73) TO ADA TRANSLATOR SYSTEM.(U)
DEC 80 R L BROZOVIC
UNCLASSIFIED AFIT/GCS/EE/80D-5 NL

Appendix E

Example Programs

J73 Version

```
START PROGRAM TSTPGM;
BEGIN
  ITEM TEST'NAME C 3;
  ITEM TEST'FLAG B 1;
  ITEM TEST'CHAR C;
  ITEM TEST'NUMBER U 3;
  ITEM TEST'NUM S;

HERE:
  TEST'NUMBER = 3;
  CASE TEST'NUMBER;
    BEGIN
      (DEFAULT) : TEST'FLAG = FALSE;
      (1)       : TEST'FLAG = TRUE;
      (2:3,5)   : TEST'FLAG = TRUE;
    END
  WHILE TEST'NUMBER = 3;
    TESTPROC;;
  IF TEST'NUMBER = 5;
     GOTO HERE;

PROC TESTPROC;
  IF TEST'FLAG;
     BEGIN
       TEST'NAME = 'ABC';
       TEST'NUMBER = 5;
     END
END
TERM
```

Ada Version

```
PROCEDURE TSTPGM IS
  TEST_NAME : STRING ( 1 .. 3 ) ;
  TEST_FLAG :
--------------------------WARNING-----------------------------
--                                                           --
--    THE CODE BETWEEN THIS BRACKET AND THE FOLLOWING        --
--          BRACKET MAY NOT BE FULLY TRANSLATED              --
--                                                           --

  B 1

--                                                           --
--    THE CODE BETWEEN THIS BRACKET AND THE PREVIOUS         --
--          BRACKET MAY NOT BE FULLY TRANSLATED              --
--                                                           --
--------------------------WARNING-----------------------------
  ;
  TEST_CHAR : STRING ( 1 .. 1 ) ;
  TEST_NUMBER : INTEGER RANGE 0 .. 2 ** ( 3 ) - 1 ;
  TEST_NUM : INTEGER ;
  PROCEDURE TESTPROC IS
    BEGIN
      IF TEST_FLAG THEN
        TEST_NAME := "ABC" ;
        TEST_NUMBER := 5 ;
      END IF ;
    END ;
  BEGIN
    << HERE >> TEST_NUMBER := 3 ;
    CASE TEST_NUMBER IS
      WHEN OTHERS => TEST_FLAG := FALSE ;
      WHEN 1 => TEST_FLAG := TRUE ;
      WHEN 2 .. 3 | 5 => TEST_FLAG := TRUE ;
    END CASE ;
    WHILE TEST_NUMBER = 3 LOOP
      TESTPROC ;
    END LOOP ;
    NULL ;
    IF TEST_NUMBER = 5 THEN
      GOTO HERE ;
    END IF ;
  END ;
```

Vita

Richard L. Brozovic was born into an Air Force family on 26 September 1953 in Anchorage, Alaska. He graduated from Thomas Jefferson High School of San Antonio, Texas in May 1972 and received a Bachelor of Science Degree in Computer Science from the United States Air Force Academy at Colorado Springs, Colorado on 2 June 1976. He was then assigned as the Data Automation Chief at Laughlin AFB, Texas. In June 1979 he entered the School of Engineering, Air Force Institute of Technology to pursue a graduate degree in computer systems.

Permanent address: 3903 Maxine Drive
San Antonio, Texas
78228

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/GCS/EE/80D-5	2. GOVT ACCESSION NO. AD-A100881	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) JOVIAL(J73) TO ADA TRANSLATOR SYSTEM		5. TYPE OF REPORT & PERIOD COVERED MS THESIS
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Richard L. Brozovic Capt USAF		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Air Force Institute of Technology (AFIT/EN) Wright-Patterson AFB, OH 45433		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Avionics Laboratory (AFWAL/AAAF-2) Wright-Patterson AFB, OH 45433		12. REPORT DATE December 1980 13. NUMBER OF PAGES 90
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

Approved for public release; IAW AFR 190-17

Frederic C. Lynch, Major, USAF
Director of Information

16 JUN 1981

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Language Translation
Grammars
Computer Programs
Programming Languages

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

In recent years the Air Force has developed a standard programming language, J73, for use in embedded computer systems. Now that the Air Force has a considerable investment in systems that are currently being developed with J73, the Department of Defense has selected a high order programming language, Ada, that will become the standard for programming embedded computer systems throughout the Department of Defense. Also under development is the definition of a support environment for Ada. One of the tools of this support environment

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

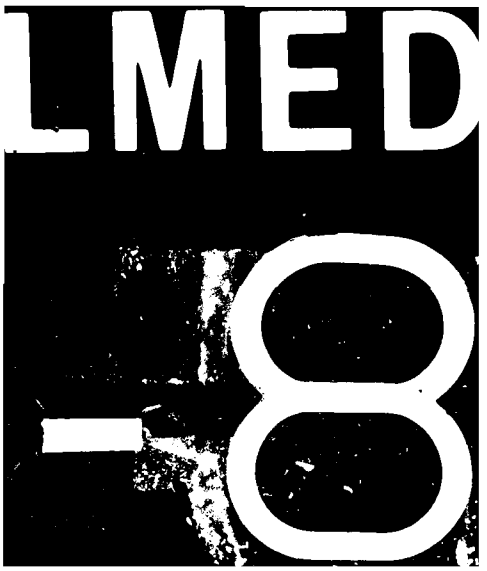
UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. Abstract(continued)

should be a translator that will produce Ada source programs from J73 source programs.

The subject of this research project was the design and development of a translation system. The resulting system accepts J73 programs and produces equivalent Ada programs to the extent possible while identifying segments of J73 code that were not translated. This project made use of language parsing techniques and various data structures to support the translation process. Several problem areas are identified with possible solutions. The translator system should be a useful tool in the transition from J73 to Ada.


LMED
-8